HUMBOLDT-UNIVERSITÄT ZU BERLIN

Model-based Testing of Dynamic Component Systems

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von
Dipl.-Inf. (FH) Siamak Haschemi

Präsident der Humboldt-Universität zu Berlin

Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II

Prof. Dr. Elmar Kulke

Gutachter:

1. Prof. Dr. Joachim Fischer

2. Prof. Dr. Bernd-Holger Schlingloff

3. Prof. Dr. Angelo Gargantini

Tag der Verteidigung: 14.07.2015

Zusammenfassung

Die Arbeit widmet sich der Frage, ob sich die etablierte Technik des modellbasierten Testens (MBT) auf eine spezielle Art von Software-Komponentensystemen, den dynamischen Komponentensystemen (DCS), anwenden lässt. DCS bieten die besondere Eigenschaft, dass sich die Komposition der Komponenteninstanzen zur Laufzeit ändern kann, da in solchen Systemen jede Komponenteninstanz einen Lebenszyklus aufweist. Damit ist es möglich, im laufenden Betrieb einzelne Komponenten im Softwaresystem zu aktualisieren oder dem System neue hinzuzufügen. Derartige Eingriffe führen dazu, dass die von den Komponenteninstanzen bereitgestellte Funktionalität jederzeit eingeschränkt oder unverfügbar werden kann. Diese Eigenschaft der DCS macht die Entwicklung von Komponenten schwierig, da diese in ihrem potentiellen Verhalten darauf vorbereitet werden müssen, dass die von ihnen jeweils benötigte und genutzte Funktionalität nicht ständig verfügbar ist.

Ziel dieser Dissertation ist es nun, einen systematischen Testansatz zu entwickeln, der es erlaubt, bereits während der Entwicklung von DCS-Komponenten Toleranzaussagen bzgl. ihrer dynamischen Verfügbarkeit treffen zu können. Untersucht wird, inwieweit bestehende MBT-Ansätze bei entsprechender Anpassung für den neuen Testansatz übernommen werden können. Beim MBT werden ausgehend von formalen Struktur- und Verhaltensmodellen der Komponenten, den sog. Testmodellen, mit Hilfe heuristischer Auswahlverfahren automatisiert Testfälle generiert. Um ihre Anwendung auch für DCS zu ermöglichen, wird zunächst eine Formalisierung der Artefakte und Prozesse des MBT vorgenommen, wobei modellbasierte Technologien zum Einsatz kommen, von den Eigenschaften des Zielsystems und der Modellierungssprache der Testmodelle jedoch abstrahiert wird. Im Anschluss wird diese Formalisierung für reaktive, ereignisbasierte, potenziell asynchron kommunizierende und zeitlose Komponentensysteme, sowie für die Unified Modeling Language (UML) als Modellierungssprache der Testmodelle verfeinert. Die Testmodelle werden mit den Klassen- und Zustandsdiagrammen der UML erstellt und als Testfallgener-

ierungstechnologie wird ein Ansatz auf Basis von Model Checking gewählt. Der letzte Schritt, um die Anwendung von MBT für DCS zu ermöglichen, wird anhand einer speziellen Struktur der Testmodelle erreicht, die den Lebenszyklus von DCS-Komponenten im Testmodell nachbildet. Damit wird es möglich Testfälle zu erzeugen, die das Zielsystem dazu stimulieren, die Komponenteninstanzen in verschiedene Phasen des Lebenszyklus zu versetzen, und damit auch eine dynamische Verfügbarkeit von Funktionalität im System zu erreichen. Insbesondere ergibt sich für die Gesamtfrage der Dissertation, dass die Anwendung von MBT für DCS keine weitere Verfeinerung der eingesetzten Modellierungssprache zur Erstellung von Testmodellen benötigt.

Durch die in der Dissertation entwickelten Ansätze sowie deren Implementierung und Anwendung in einer Fallstudie wird gezeigt, dass eine systematische Testfallgenerierung für dynamische Komponentensysteme mit Hilfe der Anwendung und Anpassung von modellbasierten Testtechnologien erreicht werden kann.

Abstract

This dissertation devotes to the question whether the established technique of model based testing (MBT) can be applied to a special type of software component systems called dynamic component systems (DCSs). DCSs have the special characteristic that they support the change of component instance compositions during runtime of the system. In these systems, each component instance exhibits an own lifecycle. This makes it possible to update existing, or add new components to the system, while it is running. Such changes cause that functionality provided by the component instances may become restricted or unavailable at any time. This characteristic of DCSs makes the development of components difficult because required and used functionality is not available all the time.

The goal of this dissertation is to develop a systematic testing approach which allows to test a component's tolerance to dynamic availability during development time. We analyze, to what extend existing MBT approaches can be reused or adapted. In MBT, test cases are generated using formal structural and behavioral models, the so called test models, and heuristic test selection methods. To enable the application of MBT to DCSs, we first formalize the artifacts and processes of MBT. This formalization uses model-based technologies and abstracts some properties of the system under test (SUT) away. Next, we refine this formalization for reactive, event-based, potentially asynchronous communicating and timeless component systems. For the test models, we use the Unified Modeling Language (UML), in special, UML class diagrams and UML Statecharts. For test generation we use an model checking approach. The last step to allow application of MBT to DCSs is achieved by a special structure for the test modes, where the lifecycle of dynamic components is imitated. With this approach, it is possible to generate test cases which stimulate the SUT to bring component instances to desired phases of their lifecycle, and therefore reach dynamic availability in the system. For the question of the dissertation, we conclude that the application of MBT for DCSs does not need any refinement for the used modeling languages for creating

test models.
The approaches of this dissertation has been implemented in a software prototype. This prototype has been used in a case study and it has been showed, that systematic test generation for DCSs can be done with the help of MBT.

Contents

CHAPTER 1

Introduction

1.1. Motivation

Modern software systems are developed using software components as the main building blocks for self-contained and reusable code. Component-based software development aims to reduce the cost of creating increasing large and complex software systems. Like in traditional engineering disciplines, the vision is to create a software system by assembling pre-fabricated parts, in contrast to the traditional way of building software custom-designed from the scratch (Gross, 2004). The hope is to cope with the problem of increasing requirements of today's software systems by maintaining the software quality by reuse of well-tested code.

As with any other software, a component-based system will go through some kind of *evolution* over its lifetime. Evolution is needed to fix bugs or implement new requirements. The traditional way to evolve a system with new functionality, or provide fixes for existing functionality, has required recompiling components and creating new configurations, or at least stopping and starting the component platform. However, in high available and safety critical systems, it is costly or risky to stop and restart a system in order to change its configuration (Oreizy et al., 1998). A number of attempts to solve these problems at the level of evolving components have been made (Kramer and Magee, 1990; Eisenbach et al., 2002; Dashofy et al., 2002; Klus et al., 2007).

In this dissertation, we focus on a special kind of component systems, called *dynamic component systems* (DCS). DCSs allow reconfiguration of the overall system during *runtime* by defining a component lifecycle for each component instance. In these systems, compo-

nent instances can be stopped at any time, which causes their provided functionality to be withdrawn. When the provided functionality was used by another component instance, this leads to the problem of *dynamic availability* of functionality.

1.2. Problem Statement

An important aspect of *dynamic availability* is that it is not under the control of the component instance using the functionality. Component instances therefore have to be prepared to respond at any time to arrival and/or departure of required functionality (Cervantes and Hall, 2003). This poses additional quality requirements to the development of software components, since missing or incorrect handling of dynamic availability can have impact on the functionality of the overall system. To cope with this problem, in this dissertation we analyze a systematic and automated way of *testing* a component's tolerance to dynamic availability during development time.

1.3. Approach

One approach to systematic testing is model-based testing (MBT). Its basic process is shown in Fig. 1.1. In MBT, test cases are generated using structural and behavioral models, the so called *test models*, and *test selection criteria*. The test model is created based on informal requirements to the system. Test selection criteria are heuristic methods to narrow down the potentially infinite number of test cases needed to completely test the system. The MBT approach has been applied to many types of systems, like realtime, continuous, reactive, and nondeterministic systems (Larsen et al., 2005; Broy et al., 2005; Fraser and Wotawa, 2007a). To take the specifics of these systems into account, several test generation approaches have been proposed and different modeling paradigms and notations (i.e. state-based notations like Z (Woodcock and Davies, 1996) and transition-based notations like Statecharts (Harel and Politi, 1998)) have been used to create behavior models for the purpose of testing.

1.4. Hypothesis

Dynamic component systems are a promising development approach for high available and safety critical systems, where stopping the system is not possible or too costly. However, developing dynamic components poses new challenges for developers, since they have to

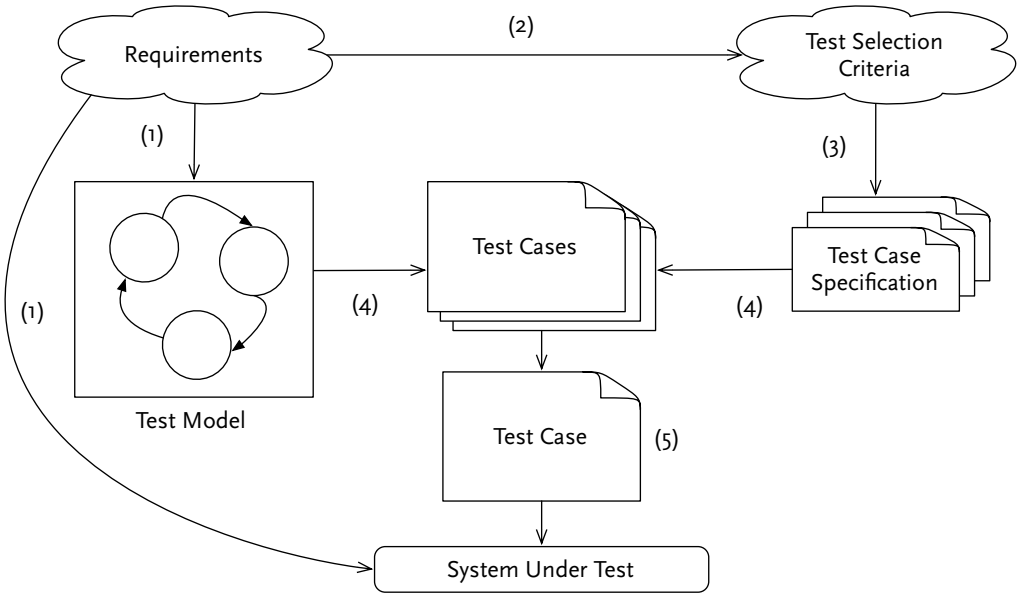


Figure 1.1.: The process of model-based testing. Based on drawing in (Utting et al., 2012).

prepare their components for dynamic availability. Model based testing is a well-established testing technology which allows systematic testing of software systems. The hypothesis in this dissertation is therefore that:

> *Model based testing can be applied to systematically test Dynamic Component Systems.*

1.5. Contributions

To show this hypothesis, we contribute in this dissertation the following:

- We formalize MBT with the aim of automatic execution of test generation using *metamodel* and *workflow* technologies. In detail, we contribute:
  - a common metamodel for MBT to describe the artifacts and concepts involved in test generation and
  - an approach to formalize the process of MBT using *abstract* workflows.
- We refine this formalization for reactive component systems using UML. In detail, we contribute:

    - an UML based test model using class diagrams for describing the structural part, and Statecharts for the behavioral part.
    - a metamodel-based formalization of test cases, test steps, test inputs, and test outputs aligned with our UML based test models.
    - four structural test selection criteria in form of metamodel descriptions, and model transformations for their semantics.
    - an approach to express additional test case specifications using LTL expression which access the UML based test model structure.
    - a workflow for automatic test generation using model checkers, and show by example that the model checker NuSMV can be used within our framework to generate test cases.
- We apply our approach and test generation tooling to generate test cases for DCSs. In detail, we contribute:
    - a novel approach to structure UML based test models to be able to generate test cases for DCSs.
    - test selection criteria with the focus on generating test cases for dynamic availability
- We finally show the applicability of our approach using a case study of Self-Organizing Seismic Early Warning Information Network (SOSEWIN) program.

1.6. Organization

This dissertation is structured as follows (Fig. 1.2):

- In Chap. 2, we present preliminaries of this dissertation.
- In Chap. 3, we present an approach to formalize the structural and behavioral parts of MBT using metamodels and abstract workflows.
- In Chap. 4, we refine the previous formalization for component systems using the UML notation.
- In Chap. 5, we show how our refined formalization can be used to systematically generate test cases for DCSs.

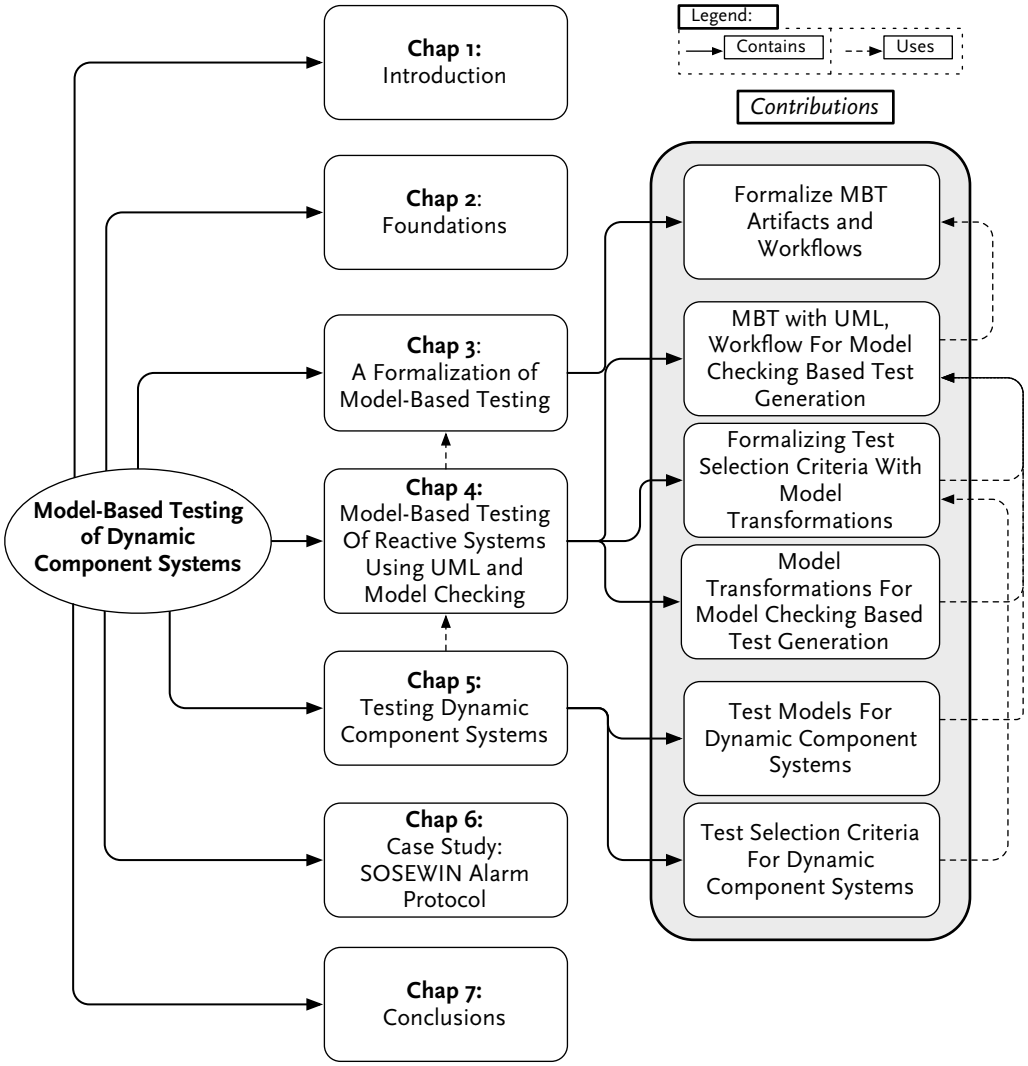


Figure 1.2.: Overview of the chapters and contributions of this dissertation.

- In Chap. 6, we present a case study based one parts of the SOSEWIN alarm protocol. In this case study, we use the approaches and tools developed in this dissertation.
- We conclude the dissertation in Chap. 7.

CHAPTER 2

Foundations

2.1. Component-based Software Development

Component-based software development aims to reduce the cost of creating increasing large and complex software systems. Like in traditional engineering disciplines, the vision is to create a software system by assembling pre-fabricated parts, in contrast to the traditional way of building software custom-designed from the scratch (Gross, 2004). The hope is to cope with the problem of increasing requirements of today's software systems by maintaining the software quality by reuse of well-tested code. The building block of component-based software development is a *software component* (also simply called *component* in this thesis). The benefits of software components have first been explained by McIlroy in 1968. In his work, software components provide "routines to be widely applicable to different machines and users" (McIlroy, 1968). Nowadays, this approach has been accepted for building cost-effective and flexible software systems. However, the term *component* is treated differently in available component technologies, which can lead to confusion. A number of attempts have been made to define what software components are, and what characteristics they have. These definitions are sometimes oppositional. For example, while (Sametinger, 1997) allows components to be available as source code, others like (Jacobson et al., 1992) see a component as binary building blocks. One of the well-known definition from the 1996 European Conference on Object-Oriented Programming (ECOOP) defines a component in the following way:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be

> deployed independently and is subject to composition by third parties (Szyperski, 1997).

However, many other definitions that focus on similar properties of a component have been proposed:

> A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard (Heineman and Councill, 2001).

> An independently deliverable piece of functionality providing access to its services through interfaces (Brown, 2000).

With respect to these definitions, a number of component properties can be derived according to (Gross, 2004; Brown, 2000; Heineman and Councill, 2001; Szyperski, 1997):

- Components can be combined and deployed to build compositions or applications. Deployment covers addition, replacement, or deletion of components. The resulting composition or application is therefore not monolithic.
- Reusability is one of the fundamental concepts of components. They are designed and built for reuse by component providers, and are used and assembled by component developers to create compositions or applications.
- Components have a unique identity, which makes it possible to identify it in the development and deployment environment.
- A component is built by principles of encapsulation and information hiding. Access to the functionality of a component is therefore required and done through external interfaces.

Component Model

The *component model* describes a set of rules and standards for the implementation and deployment of components. It describes in particular the component interfaces. The component interface gives access to syntax and semantics of a component through *provided* and *required* interfaces. The provided interface is a collection of functionality that a component provides to its clients. Clients can control the component and use its functionality through this interface by interacting with it. With the required interfaces, the component expresses

requirements to the environment in order to provide its own functionality. In general, a component will have multiple interfaces corresponding to different, separated functionality (Szyperski, 1997).

With the component's interfaces, it is possible to fulfill the most important concept of components, namely composition and assembly. The fundamental concept behind the component composition is the client/server relationship or *clientship* (Gross, 2004). This concepts, borrowed from object technology, represents the basic interaction between two components. Clientship represents a communication between a client instance and a server instance. This communication is unidirectional, so that the client instance does have knowledge about the server instance (typically a reference value), but the server instance has no knowledge about the client instance. The clientship represents a contract between client and server, so that the server promises to provide its functionality if the client uses the server in a expected way. The interaction fails if one of the two fails to fulfill the contract. With the clientship relation, component compositions can build arbitrary graphs (Gross, 2004). This is a more relaxed definition than the *composition* definition, where the composite means containment. Figure 2.1 shows a Unified Modeling Language (UML) like illustration of the above concepts. While the clientship relation forms a contract for the interaction of com-

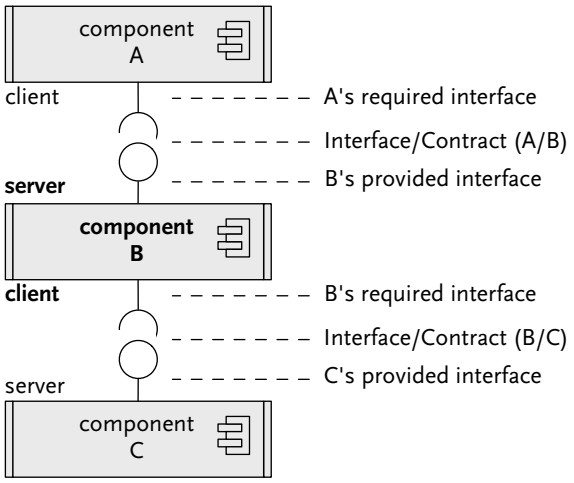


Figure 2.1.: Representation of components with provided and required interfaces. Based on illustration in (Gross, 2004).

ponents, it does not detail the interaction primitives for components. These primitives describe how two components can communicate during runtime. As with general distributed

systems, a component may communicate through various approaches. These approaches spread from remote-procedure-calls, to message/event- or stream-oriented communication. In addition, the communication may by synchronous or asynchronous. The reader should refer to (Alda et al., 2003) for an overview of these variants and in which component models they have already been adopted. The most recognized and used component models today are Enterprise Java Beans (EJB) (Oracle, 2013), the CORBA Component Model (CCM) (Object Management Group, 2006a), and Microsoft's Component Object Model (COM) (Microsoft, 2013). These component models target different, but overlapping applications areas, ranging from client to server applications. For the sake of brevity, a detailed description and comparison of these component models is omitted, but can be found in (Szyperski, 1997).

2.1.1. Component Platform

While the component model describes a set of rules and standards for the implementation and deployment of components, the *component platform* implements this component model. The component platform is responsible for allow assembling and deployment of components. *Deployment* covers addition, replacement, or deletion of components. When adding a component (often referred to *installing a component*), the component platform gets access to the component's provided and required interface description, and its implementation. When the component platform is started, it instantiates all components if all required interfaces of all components can be satisfied. We call such a instantiated component an *component instance*. Depending on the implementation detail of the component, this instantiation may result in allocating memory, loading code instructions into the memory, or any other technique to make the component instance provide its functionality. After creating the component instances, the component platform is responsible for *binding* the component instances by matching required and provided interfaces. A successful binding describes a valid *configuration* (or composite) of the system, where each component instance can provide its functionality. Figure 2.2 shows an illustration of the above steps.

2.1.2. Dynamic Component Model

Component models may support systems where the configuration of components *evolves* during *runtime*. Such systems, we call them *dynamic*, allow long-running applications, where future requirements can be met and implemented without having to stop the system. The traditional way to evolve a system with new functionality, or provide fixes for existing

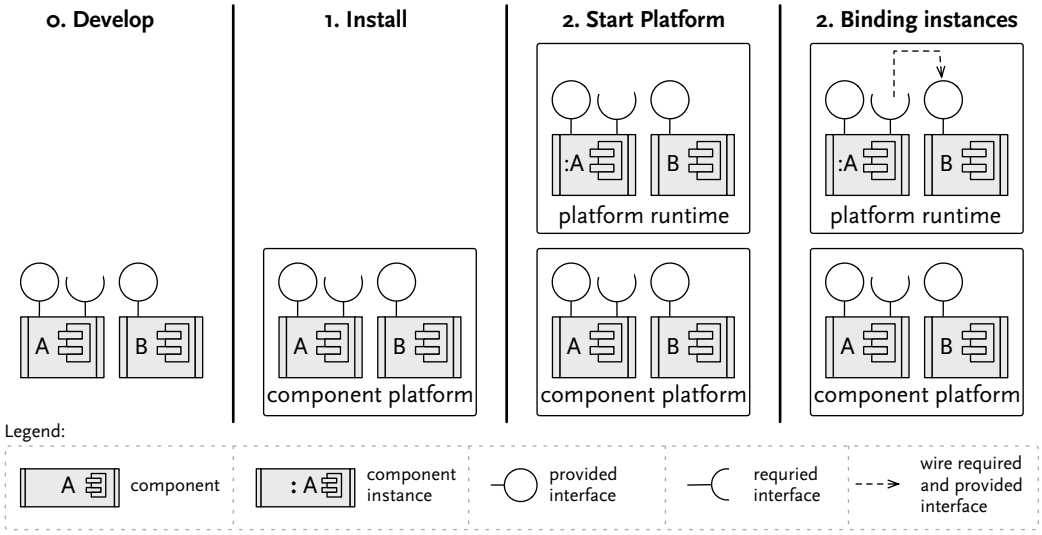


Figure 2.2.: Actions on a component for a component platform. The component is first developed in a typical development process. Then, it is installed into the component platform. Starting the component results in a component instance. When all required functionalities of the component are satisfied, the component instance is wired to component instances providing these functionalities.

functionality, has required recompiling components and creating new configurations, or at least stopping and starting the component platform. However, in high available and safety critical systems, it is costly or risky to stop and restart a system in order to change its configuration (Oreizy et al., 1998). However, changing and new requirements not known at the initial design time still may need an evolution of the system. A number of attempts to solving these problems at the level of evolving components have been made (Kramer and Magee, 1990; Eisenbach et al., 2002; Dashofy et al., 2002; Klus et al., 2007). In order to allow runtime evolution, a component model has to have two fundamental properties ((Krakowiak, 2007)):

- *Adaption of a component instance*: A component instance should be able to adapt its interfaces by withdrawing its provided functionality. Under certain circumstances (due to missing required resources), it may be impossible for a component instance to provide a functionality. In this case, the component instance needs a way to signalize this to the component platform, which may result in a new overall configuration.

- *Reconfiguration of the overall system*: On the composition level, it should be possible to reconfigure the system. Examples are changing the binding between component instances, stopping or replacing component instances, as well as adding or removing components. If any of these runtime changes happen, component instances may need to be informed about the change by the component platform.

Component Lifecycle

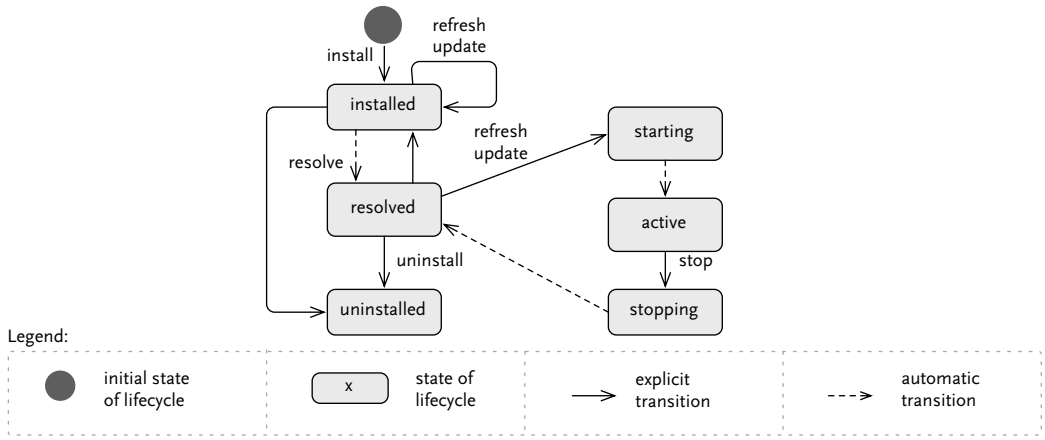


Figure 2.3.: The lifecycle of an OSGi component with explicit and automatic transitions.

To support the above property of reconfiguration of the overall system, a component model has to support a component *lifecycle*. This lifecycle describes the evolution of a component from its creation to deletion. As a representative of a dynamic component model, Fig. 2.3 illustrates the component lifecycle of the OSGi platform. To relate the explanation of the lifecycle to a generic view of an abstract component model, we will skip any unnecessary technical detail and refer to the OSGi specification (OSGi Alliance, 2007). Other existing component system propose variations and extensions of this scheme.

In OSGi, a component (also called *bundle*) is installed by loading a JAR file to represent the component in the component platform. If the contents of the JAR file are valid, the component is in the INSTALLED state. An installed component may enter the RESOLVED state when all its required interfaces are satisfied. Resolution (or binding) of a component depends on a number of constraints, mainly exported and imported Java packages with matching attributes like version number ranges, symbolic names, etc. If a component can be resolved, the contents of the JAR file are instantiated, and in terms of a component

model a component instance is created. From the resolved state, a component instance can go through the lifecycle loop of STARTING, ACTIVE, and STOPPING. When a component instance is started, a special class inside the JAR file is instantiated and a start method is called, which allows the component instance to initialize and allocate more resources. Analogous, stopping a component instance results in a call of a stop method of the class, so that the component instance can release any acquired resource. An *update* of a component forces the component instance to stop, the code of the component to be replaced by the contents of a new JAR file, resolution of the component, and the start of the component instance (if resolution was successful). The final state of a component is UNINSTALLED. From this state, no further transition is possible. The JAR file of the component is removed from the local file system. Some of the above transitions are triggered by manual actions (i.e. install, start, stop, etc.) made by a client. This client may be a real user or a software. The latter allows self-adaption of the system, where some managing component instances can start and stop other instances.

Component Interaction

To support the adaption of a component instance, a component model needs to allow component instances to dynamically provide *and* withdraw their provided functionality. Again, as a representative of a dynamic component model, we use OSGi to show how this requirement can be met. Other existing component system propose variations and extensions of this scheme. In OSGi, the adaption of components is supported by a programming methodology called *service-oriented programming* (SOP) (Cervantes and Hall, 2003). In SOP, a service is a contract of defined behavior. In this methodology, a client is not tied to a service provider, since service providers are interchangeable. SOP follows a publish-subscribe pattern where service providers publish services at runtime into a *service registry*, and clients discover these services. The uniqueness of this programming approach is that services are dynamic by definition, meaning that a service provider can withdraw a provided service at any time, and clients have to be prepared to cope with this situation. In addition, several providers can publish the same type of service, and clients need to cope with service selection where multiple services match. In OSGi, the service contract is defined by a Java interface, and service implementations are Java classes implementing this interface.

Figure 2.4 shows an exemplary interaction between two component instances following the SOP methodology. First, component instance A subscribes to the service $Is1$, mean-

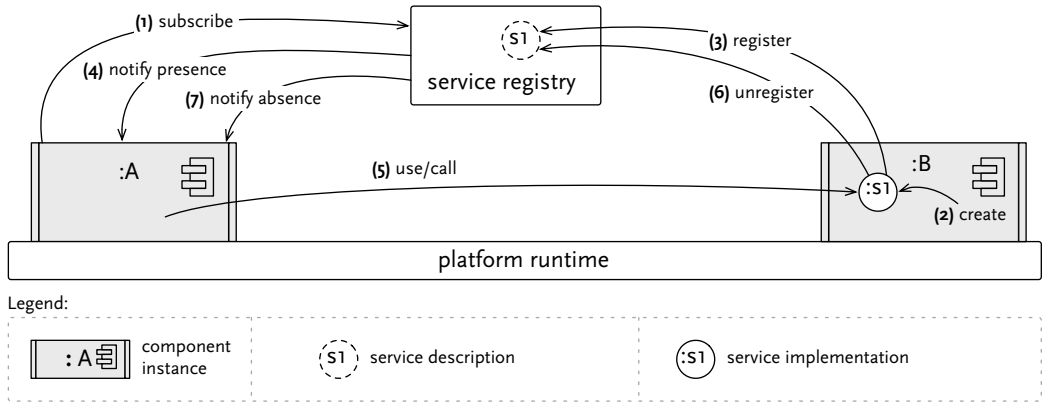


Figure 2.4.: Exemplary interaction of two OSGi component instances using the OSGi service registry for communication.

ing that it wants to be informed if any implementation of this service is available. Next, component instance B instantiates the class S_1 (which implements the Java interface $Is1$) and registers this instance in the service registry. The service registry notifies component instance A about the new instance, and A is now able to use the service. At some point, B decides to withdraw/unregister its service and informs the service registry about this situation. The service registry now notifies A about the absence of the service, and A has to cope with this situation.

2.2. Software Testing

With the increasing number of software used in all kind of industries like telecommunications, automotive and flight control, the question if the software we built does what it is supposed to do becomes very important. In mission critical systems like in the automotive area, a software failure can lead to casualties or death when software contain errors. To cope with this problem, *software testing* has become an important part of the software development process. Formal proof methods, often seen as an alternative to software testing, can show the correctness of a system, but their applicability is limited to small systems, and full automation is not possible (Fraser, 2007). Software testing, in contrast, can be automated and applied to fairly large (sub-) systems. However, with testing it is not possible to proof correctness, but only increase the confidence of the correctness of our system. If testing would be done exhaustive, test would be able to show correctness. However, in practice,

exhaustive testing is not feasible due to limited resources (budget, time, hardware). In spite of these limitations, software testing has become an important part of the software development process by providing a way to practically ensure high (or high enough) software quality with limited resources.

2.2.1. Software Testing Basics

Software development is still a largely manual process. As with any manual work done by humans, it is likely that errors are being made. When talking about software testing, the common terms *error*, *fault*, and *failure* have to be defined, since there are many different understandings of these terms. Following well known definitions in (Ammann and Offutt, 2008), a *fault* is a static defect in the software, which may be introduced by a misunderstanding of requirements, forgetting a condition, or simply mistyping. Although the software is executed, a fault may not be executed. If the faulty instruction is executed, the fault is *activated* and may result in an *error*. An error is therefore defined as an incorrect internal state that is the manifestation of some fault. This can be a wrong value of an attribute, or a null-reference. If an error affects the observable of the system under test (SUT), we speak about a *failure*. A failure is an external, incorrect behavior with respect to the requirements or other description of the expected behavior.

Software errors exposing in faults in the SUT may have an impact on the software quality. According to the ISO/IEC 9126 norm (ISO/IEC, 2001) (revisited by ISO/IEC 25010:2011 (ISO/IEC, 2012)), software quality can be classified in this set of characteristics:

- **Functionality** - A set of attributes that bear on the existence of a set of functions and their specified properties.
- **Reliability** - A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
- **Usability** - A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.
- **Efficiency** - A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.
- **Maintainability** - A set of attributes that bear on the effort needed to make specified modifications.
- **Portability** - A set of attributes that bear on the ability of software to be transferred

from one environment to another.

To cope with errors and increase the software quality, *software quality assurance* has become an important part of the development lifecycle of software. According to ISO/IEC 8402:1995-08 (ISO/IEC, 1995), software quality assurance consists of systematic actions to increase the confidence of the correctness of an implementation according to a specification. Typical methods of software quality assurance are *static analysis* and *dynamic testing*. While the former methods does not execute the software (for example inspections and static analysis tools), the latter executes the software in its real environment (for example testing).

Testing

Testing is the activity of *executing* a system in order to detect failures (Utting and Legeard, 2006). A more detailed definition of *testing* is given by the IEEE Software Engineering Body of Knowledge (SWEBOK 2004) (Beizer, 1990):

> Software testing consists of the *dynamic* verification of the behavior of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behavior.

In the definition, italicized words are further explained:

- **Dynamic:** This means that testing always implies executing the program with specific input values to find failures in its behavior.
- **Finite:** Most real programs have typically a large number of possible inputs, loops, and other behavioral constructs, so that the possible sequence of operations is usually infinite. Testing implies a trade-off between limited resources on the one hand, and unlimited testing requirements on the other.
- **Selected:** The key challenge of testing is therefore the selection of those tests, which will most likely expose failures of the system. Selection can be done manually by test engineers, or based of heuristics in form of algorithms (like in model-based testing).
- **Expected:** Testing activities succeed if a failure of the system can be exposed. When executing the software, for the observed behavior we have to decide if it represents a failure. This is called the *oracle* problem. This problem can be coped with manual inspection, or can be automated (like in model-based testing).

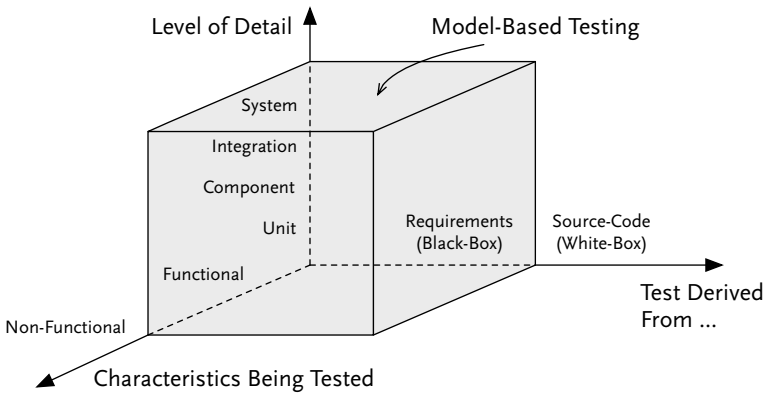


Figure 2.5.: Possible application fields of model-based testing. Model-based testing is typically used for requirements-based (black-box) testing of functional aspects of a system. It can be used to test different levels of detail (from unit to system testing). Based on drawings in (Tretmans, 2004; Utting and Legeard, 2006; Weißleder, 2010).

For each type of observable failure, related to the software quality definition, corresponding testing techniques have been proposed. Figure 2.5 shows a way to classify various kinds of testing techniques along three dimensions. One dimension shows the *level of detail* the tests target. In ranges from *unit tests*, where single methods or classes are tested, over *component tests*, where single components are tested in isolation, to *integration tests*, where several components are tested in a configuration. Finally the *system test* focus on the whole system. Another dimension shows the *characteristics* of the test technique. It separates to *functional* and *non-functional* tests. Functional tests aim at exposing functional errors of the system, i.e when for some valid input an unexpected behavior is observed. Non-functional test techniques aim at exposing other failure types like for reliability or usability. The third dimension shows from which source we build our tests. When the system requirements are the only source we use to create test cases, we call this *black-box* tests. However, if we have detailed knowledge about the structure, or even the code of the system, and create our test cases based on that information, we call this *white-box* test. In practice, typically both techniques are used. For example, even if we test every functional requirement, we may have code statements that are not executed by these tests. Then, in addition to the black-box tests, we might create some white-box tests that execute these code statements.

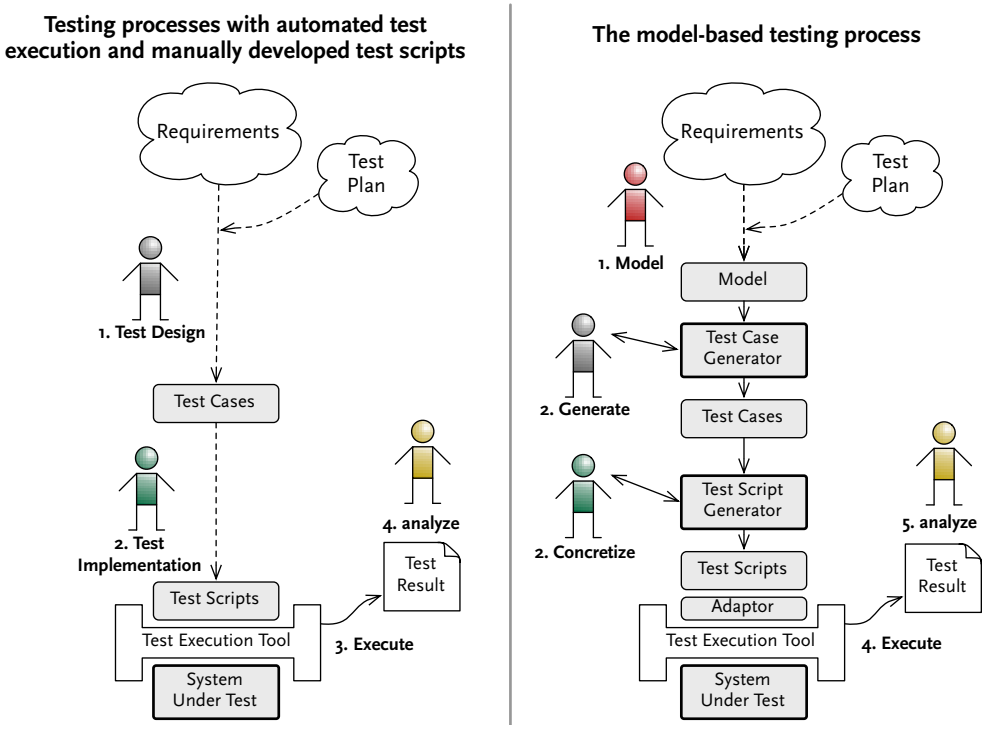


Figure 2.6.: Testing process with automated test execution and manually developed test scripts compared to the model-based testing process. Diagram is based on (Utting and Legeard, 2006).

Test Automation

The complexity of showing whether a system is implemented correctly with respect to the requirements is high. Software testing in general can be done completely manual and sporadically, and still having impact of the software quality. However, testing manually is an error prone process, and lacks of repeatable results. So the real power of testing comes with *automation*. To be able to automate testing, a systematic and traceable approach is necessary. The current practice in industry is to write manual test cases, but automate the *test execution*. Automating the generation of test cases (for example with model based testing techniques) has not been adapted widely.

Figure 2.6 (left side) shows the process of testing with automated test execution. Based on the requirements to the software, typically expressed in informal or semi-formal textual or graphical notations, a *test plan* is created. The test plan gives an overview of the testing objectives, such as which aspects of the SUT should be tested, the test strategies, how often testing should be performed, etc. (Utting and Legeard, 2006). The test plan helps to design the test cases. Designing these test cases is typically done manually, although it is time-consuming and does not ensure systematic coverage of the SUT functionality. When testing is not automated, these test cases are used by a *manual tester* to execute the software and observe its behavior. However, when test execution is automated, the SUT is started in a *test execution environment*, where its execution can be controlled, input can be injected, and behavior can be observed. This environment allows then to implemented the designed test cases and enhance them with low-level details to match the expected input of the SUT. The most important part if the test implementation is that it automatically can match the given behavior to the expected. Finally, the test results are analyzed to determine the cause of each test execution failure.

2.2.2. Model-Based Testing

Model-based testing (MBT) is an approach to automate test generation. Instead of writing hundreds of test cases manually, the test designer creates, based on the original requirements of the software, an abstract model of the SUT, and a MBT tool generates a set of test cases from this model. This procedure allows a systematic coverage of the SUT functionality, repeatable test case generation traceable based on different test selection criteria, and traceable results from the abstract model down to the executed test case (Utting and Legeard, 2006). Figure 2.6 (right side) visualizes this process. The next chapter (Chap. 3)

we present a more detailed introduction and formalization of MBT.

2.3. Model-Driven Development

In model-driven development, we use terms like *model*, *metamodel*, *model transformation*, etc. To avoid ambiguities with these words, we define the terms relevant to this dissertation in this section. Aligned with preliminary work (Harel and Rumpe, 2004; Scheidgen, 2009; Sadilek, 2010), we first define what the term *language* in the computer science literature means. A language is defined by:

- the *abstract syntax*, describing which concepts are available in a language and how they can be combined,
- the *concrete syntax*, defining how these concepts are shown to the user,
- and the *semantics*, describing what the language concepts mean.

In the computer science literature, different solutions are known for the development of software constructed with languages. For example, when we develop software with grammar-based languages, we deal with grammars and related and tools. However, in this dissertation, we focus of software constructed with *model-based* languages defined with *metamodels* and associated tools. In general, a model is a simplified *representation* of a certain real or imaginary thing. The simplification is always done for a special *purpose*, and it is typical that several representations with different purposes for the same thing exist. A model is therefore not intended to capture all the aspects of a thing, but mainly to *abstract* out only some of these characteristics. For this dissertation, we use a more specialized meaning of the term "model", aligned to (Sadilek, 2010), and assume that a model has the property that it is object-oriented data in a computer's memory. As shown in Fig. 2.7, a model has to conform to a *modeling language*. With *conformance* we mean that the model has to respect rules of the abstract syntax, the concrete syntax, and the semantics of the modeling language. The abstract syntax of a modeling language is described by a *metamodel*, so transitively, a model conforms to a metamodel. A metamodel can be understood as the representation of the class of all models expressed in that language. The metamodel itself has to conform to a *metamodeling language*, whose abstract syntax is defined by a *metametamodel*. Transitively, a metamodel conforms to a metametamodel. A metametamodel consequentially can be understood as the representation of the class of all metamodels expressed in that language. The *reflexive* metametamodel finally conforms to a language whose abstract syntax

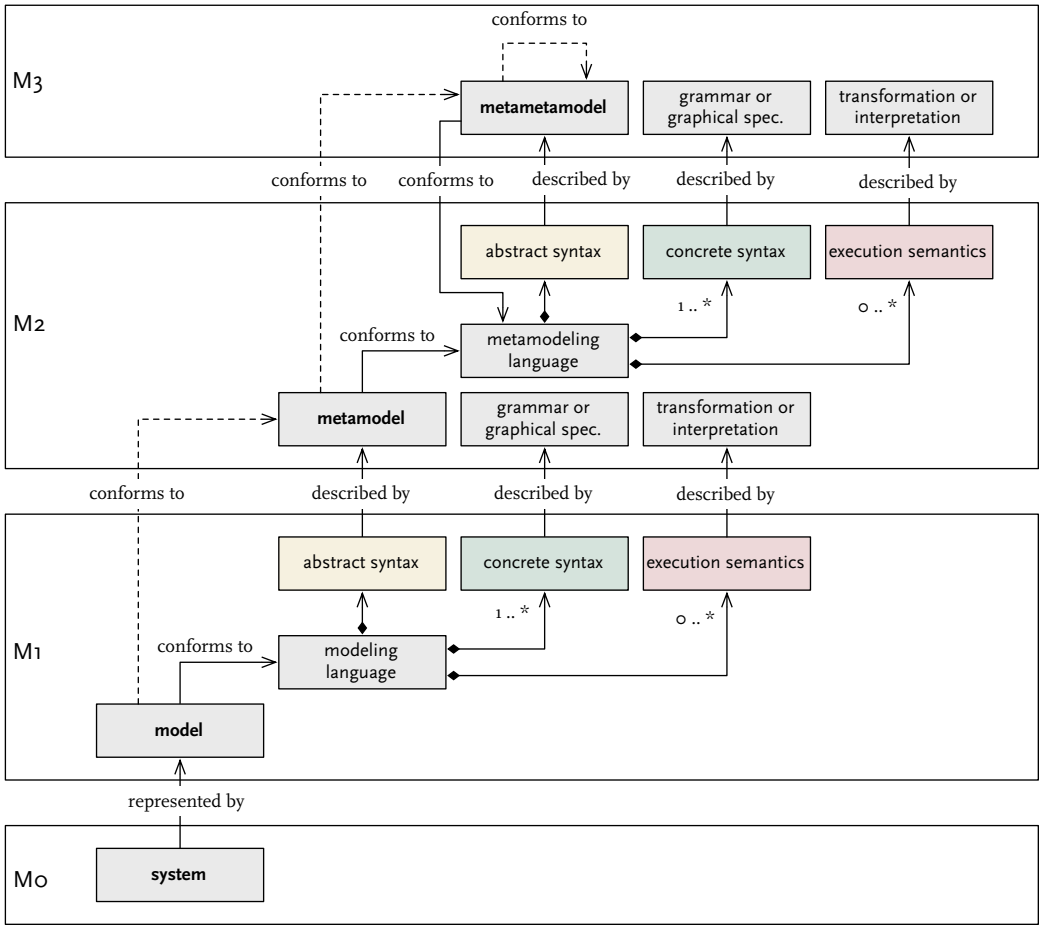


Figure 2.7.: 4-layers approach of the model driven architecture showing the system (M_0) layer, the model (M_1), metamodel (M_2), and metametamodel (M_3) layer.

is represented by *itself*. Transitively, we can say that a metametamodel conforms to itself.

This 4-layers approach originally has been proposed by the Object Management Group (OMG) in the *MetaObject Facility* (MOF) standard (Object Management Group, 2003). The reason to create the MOF was to find a language to describe allowed diagrams of the UML. Instead of inventing a new language, the designer's chose to use UML class diagrams for this purpose. These class diagrams, together with further rules written in natural language, represent the metametamodel of UML. Nowadays, MOF can be used for the development of metamodels of abritary languages. Further, it enables the MOF to "provide a more domain-specific modeling environment for defining metamodels instead of a general-purpose modeling environment" (Object Management Group, 2003). The MOF is by definition not tied to UML, and other languages can be described with it. However, the MOF does only consider the abstract syntax part in its standard. Concrete syntax definitions, or semantics, have been omitted, although recent work tried to extend it in that direction (Boronat and Meseguer, 2008; Paige et al., 2006; Mayerhofer et al., 2012; Scheidgen, 2009). To make the entrance to the development of metamodel-based tools easier for tool vendors, the MOF standard has been divided into two sub-packages, namely the Essential MOF (EMOF) and the Complete MOF (CMOF). A popular implementation of EMOF is *Ecore*, which is part of the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009).

We illustrate the 4-layer of MOF with an example of a coffee machine in Fig. 2.8. In this example, three models of a coffee machine with different purposes are shown. The first model on the left side represents the coffee machine as a UML class diagram. Another model using a UML Statechart describes the behavior of the coffee machine. Both diagram types are described using the UML metamodel. To illustrate another 4-layer approach, on the right side the behavior of the coffee machine is described using a textual notation. The abstract syntax of this textual notation is described using a metamodel based on the *Ecore* metametamodel.

Model Transformation

When developing software using metamodel-based languages, several options for describing the semantics of the language exists. Beside structure-only semantics like UML class diagrams, where no meaning for the ability to execute these diagrams exist, *executable* languages can describe what the language concepts mean. Giving a broad overview over the possible ways to describe executable semantics of languages is way beyond this dissertation,

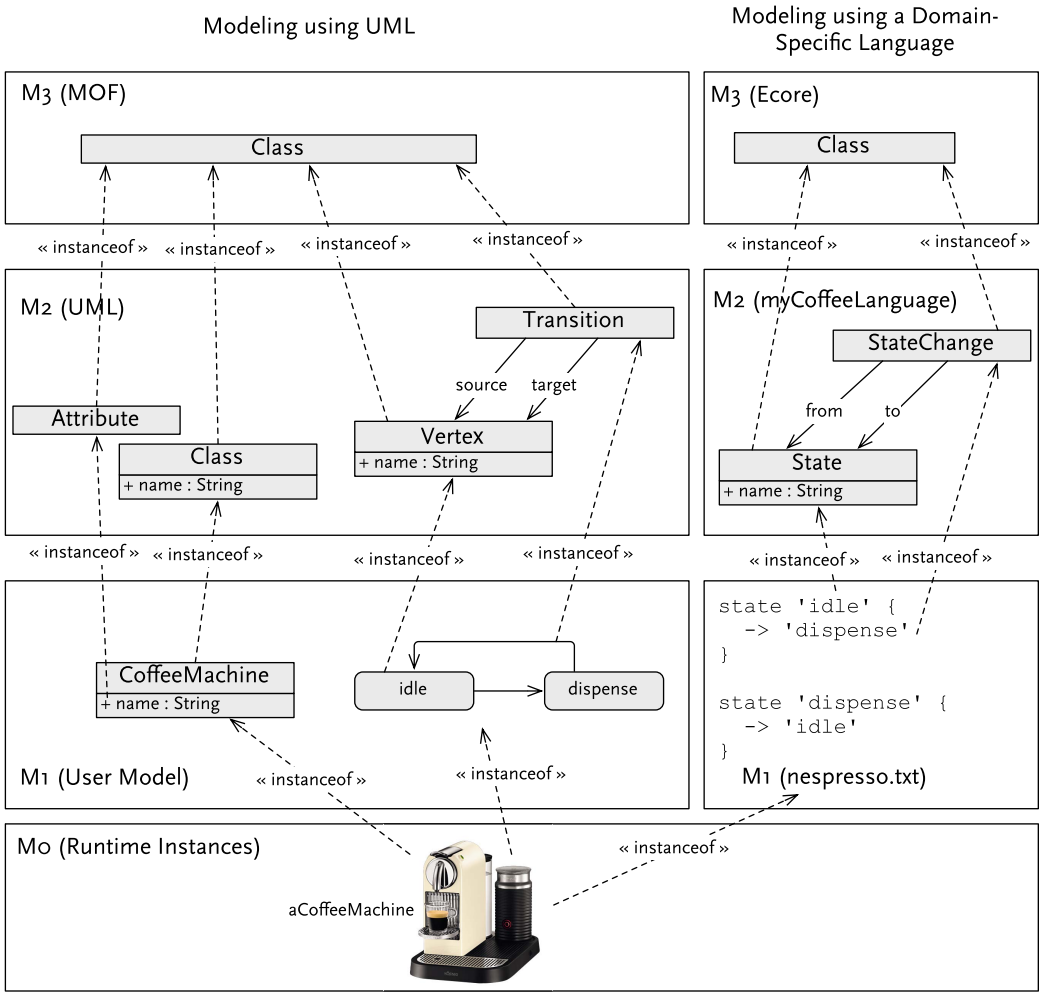


Figure 2.8.: Example of a coffee machine with three different models of it. On the left a UML class diagram describes structural aspects. In the middle a UML Statechart describes the coffee machine's behavior. In the right the behavior is described using a textual language.

and for a more detailed discussion, we refer to (Sadilek, 2010). In this dissertation, we use one special form of semantic description, namely *model transformations*. Model transformation are a means of describing a translation of a *source* language to a *target* language. A prominent usage example of model transformations is given by the OMGs *model driven architecture*, where the semantics of platform-independent models are described using a translation to platform-specific models (Object Management Group, 2009). Beside this typical usage, model transformations can be used to define the semantics of language constructs by transformation of every concepts to a language with well-known semantics.

Model transformations are distinguished between *model-to-text* (or model-to-code) and *model-to-model* transformation approaches (Czarnecki and Helsen, 2003). Model-to-text transformations translate a source model to a text. This text typically conforms to a textual programming language like Java or C++, where existing compiler technology exist. The majority of current available model-to-text transformations uses a *template*-based approach to generate text/code. Templates consist of target text and code to access the metamodel of the target language. Available tools allow programming language constructs like loops, conditional statements, and delegation to create a call order of templates. Prominent examples for these category of tools are AndroMDA (AndroMDA, 2013), JET (Java Emitter Templates (JET), 2013), and Xpand (Eclipse, 2012).

Model-to-model transformations in contrast translate between a source model and a target model. These models can conform to different, or the same metamodel. Transformation of a model to another model is typically used when a platform-independent model is refined gradually to another platform-independent model, before is gets converted to a platform-specific model. Other applications for model-to-model transformations are views of a system model and synchronization of the view and the original model (Czarnecki and Helsen, 2003). Prominent examples for these category of tools are QVT (OMG, 2011) and Xtend (Eclipse, 2012).

CHAPTER 3

A Formalization of Model-based Testing

3.1. Introduction

In this chapter we propose a formalization of MBT with the aim of automation of test generation. The formalization captures the artifacts and processes of MBT, using metamodels and abstract workflows, and is independent of the target system type and modeling notation for test models. The results are a common MBT metamodel, the idea of using abstract workflows to support families of MBT approaches, and a prototypical software implementation which shows the applicability of this approach. The formalization approach and results of this chapter are reused and refined in following chapters of this dissertation, and thus serve as the basis for showing that our hypothesis holds.

The MBT approach has been applied to many types of systems, like realtime, continuous, reactive, and nondeterministic systems (Larsen et al., 2005; Broy et al., 2005; Fraser and Wotawa, 2007a). To take the specifics of these systems into account, several test generation approaches have been proposed and different modeling paradigms and notations (i.e. state-based notations like Z (Woodcock and Davies, 1996) and transition-based notations like Statecharts (Harel and Politi, 1998)) have been used to create behavior models for the purpose of testing. A recently proposed taxonomy for MBT identifies seven dimensions to categorize these MBT approaches (Utting et al., 2012). This taxonomy shows both the diversity of existing approaches, as well as common concepts of MBT. We can group these common concepts into manually created or generated *artifacts* and the test generation and execution *processes*.

Artifacts are used to model the expected behavior of the SUT (test models), select interesting test cases with a formal criterion (test selection criteria), or describe the sequence of inputs and outputs to/from the SUT (test cases). The characteristics of these artifacts can be described on an abstract level, as we will do in this chapter. However, in order to test a specific system, we need to refine their formalization. For example, when we choose a modeling notation to create test models, i.e. one of the diagrams of the UML, the notation suggest specific kinds of test selection criteria, as identified by (Utting et al., 2012). A good example is the test selection criterion *All-States*. It requires that each state in a model has to be covered by a test case at least once. This criterion makes sense in transition-based notations, i.e. UML Statecharts, but cannot be used in state-based notations, i.e. Z (Bowen, 1998), where the system is modeled as a collection of variables and the state-space is typically huge.

The test generation and execution *processes* describe the necessary steps to perform automatic test case generation or execution. Some of the typical test generation steps are the transformation of test selection criteria into test case specifications, the ordering of test case specifications, and the post-optimization of test suites. Although there exists an agreement about the typically involved steps in a MBT generation process (Utting et al., 2012; Utting and Legeard, 2006), it is also sensible to assume that each test generation approach may require specific unique steps. For example, the process of *online testing* and *offline testing* differ, because the former executes a test case directly after its generation, while the latter first generates all test cases and executes them afterwards. Even if there is a consensus about a MBT test generation process for a certain test generation approach, there might be *alternative* approaches to choose from for every step. An example for such an alternative is the ordering of test case specifications to reduce the test suite size and test generation execution time. This step can be accomplished by different algorithms. For example, when the automatic test generation is performed using model checking (Fraser et al., 2009), one can use an approach based on subsumption relations (Hong and Ural, 2005), while other test generation approaches can use graph-based approaches (Marré and Bertolino, 1996).

In this chapter, we present an approach to formalize the MBT artifacts using *metamodels* and the test generation process of MBT using *abstract workflows*. In Sec. 3.2, we discuss the decision for metamodels and present a metamodel formalizing the common artifacts of MBT. Sec. 3.3 describes how abstract workflows can be used to formalize the workflows of MBT. We implemented our approach on top of the Eclipse platform and present this

implementation in Sec. 3.5. Finally, we discuss about the advantages and implications of the approach in Sec. 3.6 and conclude this chapter in Sec. 3.8.

3.2. Formalizing MBT Artifacts

As discussed before, there exist a variety of MBT approaches. Most of the approaches are based on a strong theory, typically described with mathematical formal languages. These formal languages make the approach independent of implementation details like programming languages. However, when we want to implement an approach, the usage of those languages poses some problems. For example, mathematical formal languages are typically not directly useable in programming languages. This might lead to an manual implementation of the approach, which can be error prone, and makes the implementations of these approaches pose a lack of interoperability. In addition, the comparison of the various formalizations of MBT created with different formal languages is limited, although they are based on common concepts. In order to formalize the MBT artifacts, we therefore need to select a formal language which allows us to create an *extensible* formalization, such that we can begin with the common concepts of MBT, and refine these concepts for a specific target system. In addition, the formal language should be independent of any programming language, but still should be easily processable and modifiable in different programming languages to avoid manual implementation of our formalization.

Based on the above characteristics, we chose a metamodel-based approach to formalize the artifacts of MBT. Metamodels satisfy the requirements we set on our formal language, namely to be extensible and interoperable. Extensibility is supported by typical object oriented features like inheritance or composition. Interoperability is supported by programming language independent representation of the metamodel and programming language independent ways to process and modify metamodels. The advantage to use metamodels over mathematical notations is that processing and manipulation frameworks for metamodels already exist. For example, we use *Ecore* metamodels in our implementation, and the *Eclipse Modeling Framework* (Steinberg et al., 2009) provides us a Java implementation of Ecore together with a rich set of tools to process and manipulate those metamodels. In addition to Java, (Sadilek and Wachsmuth, 2009) provide a framework for manipulating Ecore metamodels in other programming languages like ASMs, Prolog, and Scheme.

Using the metamodel-based approach, we formalize the common artifacts of MBT. Figure 3.1 shows our proposed metamodel. Central concepts of this metamodel are the Test-

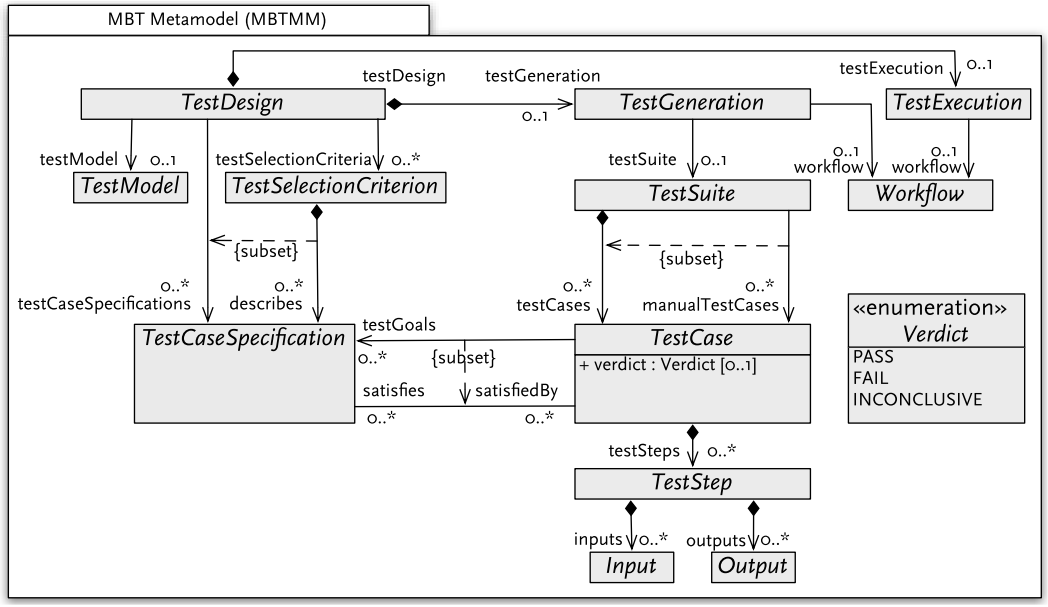


Figure 3.1.: A Metamodel covering the common concepts of MBT artifacts.

Design, the TestGeneration and the TestExecution. The TestDesign describes *what* the aim of the testing is by describing the expected behavior of the system and posing coverage requirements. The TestGeneration describes *how* these requirements can be fulfilled by a test suite. Finally, the TestExecution describes the steps used to execute the test cases contained in the test suite. All other elements in the metamodel (except the Verdict) are marked as *abstract*. This means that they cannot be instantiated and need to be concretized for the type of the target system and for the modeling notation. In the next chapter, we will present concretizations of these elements for testing component systems using UML (Chap. 4). It should also be noted that we choose to assign a lower bound of *zero* to each reference cardinality in the metamodel to allow creation of partial models.

Test Model

The expected behavior of the SUT is described with the TestModel element (Fig. 3.1). The test model is created on basis of the requirements and specifications of the SUT. For creating this model, the trade-off between the abstraction level and size of the model must be handled. The creator of a test model can raise the abstraction level of the model by focusing on specific (functional of non-functional) aspects of the SUT, which helps to reduce the efforts to create

the model compared to the complete implementation of the SUT. However, at the same time, the creator of the test model has to make sure that the model includes enough details so that the test generator is able stimulate the SUT to test the desired functionality. Creating the test model is therefore an application-specific engineering task. In the MBT context, one of the main properties of test models is to be as formal as needed to be machine processable.

Test Selection Criteria

The test model generally describes a large amount or even infinite set of behaviors. For example, when the test model uses an automaton based notation, each cycle of a loop in the model can cause an alternative behavior. If testing would be done exhaustively, a test suite covering all behaviors would guarantee correctness. However, exhaustive testing is not feasible in practice, because the number of test and the time required to generate and execute them is not manageable (Fraser, 2007). The decision which tests have to be used out of the infinite set of possibilities is difficult, and leads to the question what a "good test case" is. (Utting et al., 2012) argue that a *good test case is one that is likely to detect severe and likely failures at an acceptable cost, and that is helpful with identifying the underlying fault*. However, the authors note that this definition is not constructive and that good test cases are difficult to find. *Test selection criteria* try to approximate this notion by selecting a sub-set of behaviors of the model through a (potentially informal) description. A *test selection criterion* describes a set of test cases and can relate to different aspects of the SUT, the test model, or stochastic characterizations such as pure randomness or user profiles (Utting et al., 2012). Examples for test selection criteria are

- requirements-based criteria (functionality of the SUT),
- structural model criteria (state coverage, transition coverage),
- data coverage (boundary-value analysis),
- and random coverage.

Full *coverage* is achieved, if every item described by the test selection criterion is covered by at least one test case. Typically the coverage is given as the percentage of covered items. Test selection criteria can be used in three different ways: First, they can measure the adequacy of a test suite ("How thorough a system is exercised?"). Second, they can be used as a *stopping condition* ("Continue to generate tests until a certain coverage is achieved"). An finally, they can be used in a *prescriptive* fashion to configure the test case generator ("Try to achieve all-states coverage"). It is common practice and promises good quality test suites

to combine test selection criteria (Antoniol et al., 2002). Thus, we modeled the reference testSelectionCriteria (from TestModel to TestSelectionCriterion) in the metamodel with a *to-many* cardinality (Fig. 3.1).

Test Case Specification

While test selection criteria are intensional descriptions of sub-sets of test model behaviors ("Cover all-transitions in test model tm"), *test case specifications* are the extensional counterparts which enumerate this sub-set explicitly ("Cover transition $tm_{t1}, tm_{t2}, \ldots, tm_{tn}$"). In the metamodel, the relation describes (between TestSelectionCriterion and TestCaseSpecification) models this enumeration (Fig. 3.1). In addition to derive test case specifications using test selection criteria, it may be necessary to formulate additional, manually specified, test case specifications. These *explicit test case specifications* may be used to restrict the test case generator to use specific paths through the test model, focus the testing on common use-cases, or transform high-level requirements into test case specifications (Utting and Legeard, 2006). The set of all test case specifications (derived and explicit) is modeled in the relation testCaseSpecification (from TestDesign to TestCaseSpecification) (Fig. 3.1).

Tests

Given the test model and a set of test case specifications, a test case generator is able to generate a *test suite*. A test suite contains a set of *test cases*. Each of these test cases satisfies one or more test case specifications, and typically each test case specification is satisfied by more than one test case. In addition to automatically generated test cases, *manual test cases* can be added to the test case set. These manual test cases can be necessary when test selection criteria and explicit test case specifications are not expressive enough to cover some system behavior. A test case contains a set of *test steps*, representing the behavior of the test with *input* and *output* actions. The input set of a test step represents a stimulation of the SUT. The output set of a test step represents the expected output of the system after the input set is applied.

In order to check the conformance of the SUT to the requirements in the test model, the test cases need to be executed. The result of the execution of a single test case is represented in the Verdict element (Fig. 3.1). The verdict is build after comparing the output given by the SUT and the expected output in the test case. The verdict can have the values *pass*, *fail*, and *inconclusive*. A test *passes* if the expected output and the output given by the system are

equal, *fails* if they are different, and is *inconclusive* if this decision cannot be made. The last option becomes handy if we cannot decide whether a test passes or fails. For example, if some resource like an internet connection is required by the test case, but currently missing, we may decide to mark the result as inconclusive. Another field of application is in *non-deterministic* SUTs. In such a scenario, the SUT makes non-deterministic choices in branching points. Thus, we may recognize a difference between the output of the SUT and the expected output of the test case, although the SUT is correct. A test case might falsely detect a fault, so an inconclusive outcome would be the correct verdict in this case.

3.3. Formalizing MBT Processes

The MBT processes for test generation and execution describe the necessary steps to perform the automatic test case generation or execution. To be able to select a fitting notation to formalize these processes, we first need to characterize them: as discussed before, each test generation and execution approach may require specific steps (Sec. 3.1). However, it is sensible to assume that for some *family* of MBT approaches a common process, containing common steps, can be defined[1]. In addition, some approach belonging to a family might implement some of the common steps different than others (i.e. by using a different algorithm). In this case, we can say that the common process is *adapted*. Table 3.1 illustrates these two dimensions of a MBT process. An example for a family of MBT approaches are test generation approaches utilizing the model checking technique (Fraser et al., 2009). In Calvagna and Gargantini (2010), the authors present a test generation process for *combinatorial testing* implemented in the *ASM Test Generation Tool (ATGT)*. This process can be seen as the common process for automatic test generation using model checkers. In the paper, the authors present three alternative implementations of the *Test Predicates Ordering* task (or in our terminology *Test Case Specification Ordering Task*), a task which has major impacts on the size of the final test suite (Calvagna and Gargantini, 2010). In our terminology, three adaptions of the common step *Test Predicates Ordering* are possible.

To be able to create MBT processes as characterized above, we decided to use *workflows*. We interpret the whole test generation or execution process as a workflow, and each individual step in that process as a task. In order to support adapted processes, we introduce the notion of an adapted workflow:

[1]Note that the creation of these steps is a domain or application specific engineering task, and therefore typically not automated.

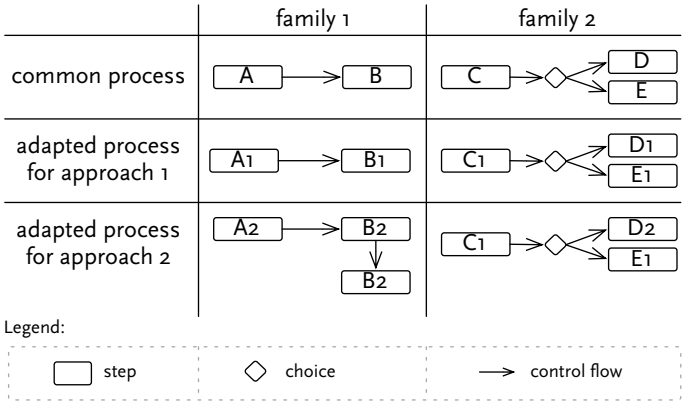


Table 3.1.: The two dimensions of creating a MBT process. Different families of MBT approaches define a suiting common process. Within a family, each approach may implement steps different (while *approach 1* of *family 1* replaces step A by step $A1$, *approach 2* replaces it by step $A2$), or add new steps (*approach 2* of *family 1* adds a step $B2$).

Definition 3.1 (Adapted Workflow). An *adapted workflow* replaces zero, one or more tasks of a workflow, can add zero, one, or more new tasks, but is not allowed to change existing control flow between tasks.

In order to support adaption of MBT processes, our approach is to model these processes as *abstract workflows*.

Definition 3.2 (Abstract Workflow). An *abstract workflow* is a workflow containing one or more *abstract tasks*. An abstract task needs to be *mapped* to a concrete (= not abstract) task to make the workflow instantiable and executable.

Abstract workflows are not instantiable and executable, until for all contained abstract tasks a *mapping* to a concrete task is defined. This mapping can be done manually (i.e. by an user), or automatically by a workflow system. In the latter case, abstract workflows design workflows at a level of abstraction above that of the target execution environment (Deelman et al., 2009). Examples for automated mapping are *service-based* and *stand-alone application* workflow systems (Deelman et al., 2009). In service-based workflow systems, mapping consists of finding and binding proper services for the abstract task. This mapping can consider a selection strategy in case of multiple matching services using a ranking function (Haschemi and Wider, 2009). In stand-alone application workflows, the mapping consists of several steps, including finding the necessary resources, performing optimizations,

and executing the stand-alone application.

We apply the concept of abstract workflows to model the MBT generation and execution processes. The general idea is to create an abstract workflow for a family of MBT approaches and use the *mapping* concept to allow the adaption of this workflow. In the mentioned example of Calvagna and Gargantini (2010), we would create an abstract workflow for *automatic test generation using model checkers* containing, beside other tasks, the abstract task *Test Predicates Ordering*. Then, this abstract task would be mapped to one of three concrete tasks representing the three alternative implementations, which makes the workflow instantiable and executable. Note that we assume an exchangeability between the alternative concrete tasks for an abstract task. The only (informal) constraint we set is that the concrete task should implement the intention of the abstract task. For example, all possible implementations of the *Test Predicate Ordering* should keep the length of the predicate set untouched, while the order of the predicates is up to the implementation. These constraints on concrete tasks could be formalized for any abstract task. However, the details of formalizing this constraints or even guaranteeing the exchangeability of concrete tasks is not part of this dissertation as left for future work.

In the MBT metamodel (Fig. 3.1), we refer to a Workflow element both from the TestGeneration and the TestExecution element, representing the generation workflow and the execution workflow, respectively. When these workflows are executed, they are provided with an instance of the MBT metamodel. The tasks within the workflows can then act on the metamodel instance, and potentially change it through model transformations. With this approach, the issue of test generation and execution is translated to a (series of) model transformations. An example for such a transformation is the creation of TestCase elements, which is is one of the results of the generation workflow. Another example is the result of the execution workflow, where Verdict elements are assigned to the test cases (Sec. 3.2).

3.4. Example

We summarize our approach with an example presented in Fig. 3.2. This example shows a oversimplified workflow to generate test cases, and should not be seen as a proposal for a real-world workflow. The workflow consists of one concrete, and three abstract tasks. For each abstract task, one or more concrete tasks are available in the system. Using a specific mapping strategy, we map each abstract task to a concrete task and thus make the workflow

executable. [2]

The abstract task Model Reading reads a MBT model (instance of the MBT metamodel, denoted as M) and makes that model available to the workflow instance. A exemplary concrete task would need to parse a concrete representation of the model, i.e. by reading a file from the hard disk. The next abstract task applies model transformations on this model to sort the test case specifications (results in M'). Here we could use one of the mentioned algorithms described in Calvagna and Gargantini (2010) as concrete tasks. The task Test Suite Generation with NuSMV Model Checker is modeled as a concrete task. It generates test cases (results in M'') by utilizing the NuSMV model checker. Finally, the abstract Model Writing task persists the transformed model M''. Here an exemplary concrete task could serialize the model into a file.

This example shows two strengths of our approach: First, we decouple the description of our MBT process (realized as an abstract workflow) from the selection of a concrete approach (by mapping abstract tasks to concrete tasks). Second, we show that mixing concrete and abstract tasks is possible.

3.5. Implementation

In this section, we present the prototypical implementation of our presented approach to formalize the artifacts and the workflows of MBT. In a nutshell, this prototype implements the MBT metamodel shown in figure 3.1 and provides a workflow system supporting the creation of abstract workflows with service-based mapping. As discussed in the last sections, in order to be able to apply our approach, a refinement of the MBT metamodel for a particular type of system and modeling notation, and the creation of an abstract workflow for a specific test generation technology is needed. The prototype is therefore not usable on its own, but only with such a refinement. In the next chapter, we will present such a refinement for component systems as the target system, UML as the modeling notation for test model, and model checking technologies for test generation (Chap. 4).

The general architecture of the prototype is shown in figure 3.3. It provides two Eclipse Plug-Ins, which rely on existing projects from the Eclipse community, namely the Eclipse Modeling Framework (EMF) and the Eclipse Workflow Engine (MWE). The first project provides tools to create metamodels and generate Java code to create and manipulate models.

[2] Note that with our formalization, we cannot argue whether all concrete tasks implement the intention of the abstract task. Thus, we cannot argue about the usefulness of the resulting workflow instance.

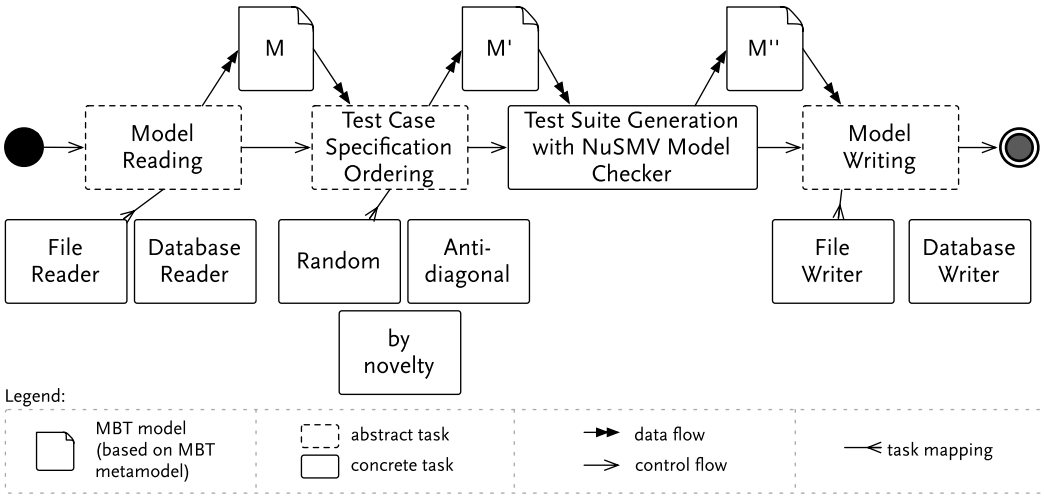


Figure 3.2.: Example MBT process to demonstrate the concepts of our approach to formalize the artifacts and the workflows of MBT. A simplified workflow to generate test cases is shown. One concrete task and the abstract tasks are defined. The abstract tasks are mapped to concrete tasks. The resulting workflow instance reads, transforms, and writes a model based on the MBT metamodel. The possible concrete tasks for the abstract task Test Case Specification Ordering are taken from Calvagna and Gargantini (2010).

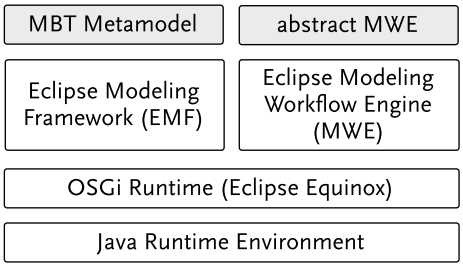


Figure 3.3.: Architecture of the prototypical implementation of the MBT metamodel (Fig. 3.1) and the workflow system supporting service-based task mapping.

The second project is a workflow system, which executes tasks implemented in Java.

Implementing the MBT metamodel

The MBT metamodel (Fig. 3.3) is implemented using Ecore, the EMOF implementation of EMF. We use model-to-text transformations provided by EMF to generate a Java API for our metamodel. With this API, we can create and manipulate models conforming to our metamodel. Note that, beside Java, other bindings can be created which allow handling Ecore-based models in other programming languages like ASMs, Prolog, and Scheme (Sadilek and Wachsmuth, 2009).

Implementing the MBT abstract workflows

In MBT, the test generation is automatically executed by abstract workflows, which do not need user intervention. Therefore, a manual task mapping strategy is not sufficient. Instead, the workflow system executing these workflows should support automatic mapping. We decided to use *service-based task mapping*, which is well supported by the underlying OSGi runtime we use for our prototype. This prototype is based on projects part of the Eclipse Platform. The Eclipse Platform uses an OSGi compliant runtime (provided by the Eclipse Equinox project) as the underlying component system. Through this system, communication between Plug-Ins is realized: A Plug-In can provide a *service instance* for a *service type* to a *service registry*. Service types can be Java classes or Java interfaces. Service instances can be any java object implementing or extending a service type. Additionally, the registration of a service instance can be attached with arbitrary non-functional properties, expressed as key-value-pairs. These properties are used to distinguish possible alternative implementations of a service type. Clients can discover service instances by querying the service registry, and filter the instances by the non-functional properties.

As a small example, imagine a printing system using OSGi, where one color and one black-white printer should be made available to users. In this system, we could represent all printers in the service type *printer*, and make two printer service instances (representing the two physical printers) available in the OSGi registry. To be able to distinguish the printers, we attach the capability (color/black-white) of each printer as non-functional properties to the service registration, i.e. $capability = color$ or $capability = blackwhite$. When a user wants to use a printer, she would need to query the OSGi registry for the *printer* service type. However, in order to specify that she needs her document to be printed on a color printer,

she could add the filter $capability = color$ to her query. The OSGi runtime would then query for all *printer* service instances matching the given filter.

For service-based task mapping, a sensible choice is to rely on the existing service registry of the OSGi runtime. In such an approach, the workflow description would model an abstract task as a service type, potentially with an extra filter for the non-functional properties. Eclipse Plug-Ins can then register concrete tasks as service instances of this service type in the service registry. Finally, the workflow system performs the task mapping by querying the service registry to map all abstract tasks to registered concrete tasks.

Unfortunately, no workflow system exists which supports the proposed approach of service-based task mapping using the OSGi service registry. We therefore use the existing workflow system MWE and extended it to support our approach. MWE is a simple workflow system where a XML dialect is used to create workflow descriptions. Workflow descriptions in MWE consist of a defined order of concrete tasks, represented by Java classes. MWE does not support parallelism and looping natively. However, the basis of MWE is Java, so the full power of the Java language can be used to implement new concepts like parallelism. We extend MWE and applied a service-based task mapping strategy using the OSGi service registry. The main extensions are the following:

- the support of abstract tasks in the workflow definition by referencing Java interfaces in addition to Java classes
- addition of XML-attributes cardinality and filter to control task mapping
- modification of the task instantiation logic of MWE to perform service-based task mapping.

The cardinality attribute allows to specify following values:

- $0..1$: Optional and unary (Default)
- $0..n$: Optional and multiple
- $1..1$: Mandatory and unary
- $1..n$: Mandatory and multiple
- where n is a natural number greater than one or $*$ (denoting unbounded multiplicity).

The cardinality attribute is introduced to support two cases in the common MBT workflow: First, it allows to define *optional* tasks. Optional tasks allow to continue the workflow execution in cases where no concrete task to map to can be found. Second, it allows to specify *multiple* possible mappings for one concrete task. To illustrate the need for multiple

mappings, imagine a common MBT workflow which contains an abstract task to generate statistics for the test generation. Further, imagine that there exist several metrics for which statistics should be created. Having the support for multiple task mapping, we can supply a concrete task for every metric. The workflow system would then map the abstract statistics task to all of our concrete tasks.

The *filter* attribute is used to filter service instances, representing concrete tasks, based on their non-functional properties, which are an optional part of the service registration. The syntax of the filter attribute is defined by the OSGi standard and based on the LDAP Search Filters (RFC 1960) (Howes, 1996).

Listing 3.1 shows the algorithm for the task mapping extension. The normal behavior of MWE is to instantiate a Java class directly (via the reflective default constructor call $class.newInstance()$). We modified this behavior of MWE to use our function, so that prior to the normal instantiation a query to the OSGi registry is made for the task type. If any service instance is found, this instance is used. Of no service instance is found in the OSGi registry, the original behavior of MWE of instantiating is used. In case the service type is a Java interface, the *null-object pattern* is used to create an interface instance where every method is empty or returns null.

Listing 3.2 shows the workflow definition for the example in Figure 3.2 using our service-based task mapping extension. The XML dialect shown is defined and interpreted by MWE. It represents every task in the workflow with the XML element component. Every child of an component element is designed to be an attribute of the underlying object, following the JavaBean convention for getters and setters. For example, the first component in the listing specifies the XML element uri with the value model.mbt. When MWE instantiates the class given for the component-element, is calls the setUri(String) method of the object with the parameter 'model.mbl'. Note that we preserved this style of passing parameters to tasks in our service-based task mapping modification of MWE.

3.6. Discussion

Metamodels

In our approach, we used metamodels to formalize the artifacts of MBT. This approach has some advantages: First, we can leverage on object-oriented extension mechanisms like *inheritance* to refine the MBT metamodel for a specific target system. This will allows us,

```
1  // this procedure performs service-based mapping of abstract tasks
2  // if no service instance can be found, it falls back to the default MWE behavior
3  serviceBasedTaskMapping(type ∈ <Class, Interface>,
4     cardinality ∈ Θ_Service-Cardinality, filter ∈ Θ_OSGi-Filter) : Object {
5
6     // try to find a service instance for the type
7     if(type == Interface) {
8        service = findService(type, cardinality, filter)
9        if(service != null) {
10          return service
11       }
12    }
13
14    // fall back to the default MWE behavior (creates instance of the type)
15    return type.newInstance()
16 }
17
18 findService(type Interface, cardinality ∈ Θ_Service-Cardinality, filter ∈ Θ_OSGi-Filter) : Object {
19    //search all registered services instances of the given type, filtered by the given osgi filter
20    List services = osgiServiceRegistry.getAllServices(type, filter)
21
22    if(cardinality.isMandatory && services.size == 0) {
23       throw Exception("Mandatory task mapping failed")
24    }
25
26    if(services.size > 1 && cardinality.isUnary) {
27       // choose an arbritary item of the list
28       return services.any()
29    } else {
30       return services
31    }
32
33    // MWE cannot handle Java Interfaces itself, so we have to instantiate a NULL-Object
34    return createNullObjectFor(type)
35 }
```

Listing 3.1: Pseudo code for the service-based task mapping extension of MWE.

```
<workflow>
    <!-- the abstract Task "Model Reading" modeled as the Java Interface "IModelReading" -->
    <component class="examples.IModelReading" cardinality="1..1">
      <uri value="model.mbt"/>
      <modelSlot value="MBTModel"/>
    </component>

    <!--
      This abstract task is marked as optional (using the cardinality attribute).
      The workflow execution continues if the abstract task cannot be mapped.
      The "filter" attribute filters potential concrete tasks.
    -->
    <component class="examples.ITestCaseSpecificationOrdering"
     cardinality="0..1" filter="(strategy=random)">
      <modelSlot value="MBTModel"/>
    </component>

    <!--
      This is a concrete task, where no mapping is needed.
    -->
    <component class="examples.TestGenerationWithModelCheckerImpl">
      <modelSlot value="MBTModel"/>
    </component>

    <component class="examples.IModelWriting" cardinality="1..1">
      <uri value="model.mbt"/>
      <modelSlot value="MBTModel"/>
    </component>
</workflow>
```

Listing 3.2: Workflow description of the example in Fig. 3.2, expressed in MWE with service-based extensions.

for example, to refine the MBT metamodel for UML Statecharts as the modeling notation for test models, and define several test selection criteria for that notation (Chap. 4). Second, metamodels are independent of programming languages or execution environments, in which metamodel instances are created and manipulated. This makes our approach applicable in other environments than supported by our prototype. Finally, the creation of a common metamodel helps in communicating, comparing, and judging an MBT approach (realized as a refinement of the metamodel).

However, our approach also leaves open issues. For example, we made a couple of design decisions while creating the MBT metamodel. This decisions are arguable and might have resulted in a metamodel which fits for our purposes only. One of these decisions was to avoid including any modeling notation specific test selection criterion in the MBT metamodel. However, some criteria like *All-States* or *All-Transitions* are often mentioned for different modeling notations (Utting and Legeard, 2006), so the decision to include some of them in the MBT metamodel would be sensible. Another design decision was that we did not include workflow related concepts in the MBT metamodel. For example, we could add elements like Task, Resource, Dependency, and further concepts from the workflow community (Hollingsworth et al., 1994). We decided that these concepts should be added by a refinement of the MBT metamodel. In summary, it is fair to say that the metamodel can only be seen as a proposal to the MBT community and might undergo changes in future.

Workflows

In our approach, we used *abstract workflows* to model the test generation workflow of MBT. The abstract tasks within this abstract workflow are mapped (using a mapping strategy) to concrete tasks. We made the assumption that some *family of MBT approaches* can be identified, and that an abstract workflow can be used to model the workflows of this family. Within a family, there might be different approaches for each task, and task mapping was our approach to cope with this problem. Our approach has several advantages: First, it makes the single steps necessary to generate or execute test cases transparent by explicitly model these steps as tasks. The resulting workflow can help in communicating, comparing, and judging an MBT approach. Second, abstract workflows allow to design workflows at a level of abstraction above that of the target execution environment. This approach makes the workflow extensible by definition. We think that this approach can lead to a set of common workflows for different families of MBT approaches. Finally, our approach allows to

treat the issue of test generation as a series of model transformations. We achieved this by using models (based on our MBT metamodel) as data in the workflow. The tasks can then implement their functionality by transforming this models.

Our approach has also some implications we like to discuss: First, similar to the issues with the MBT metamodel, we made design decisions which might lead to a approach fitting our purposes only. Second, it might turn out that our assumption about the existence of *family of MBT approaches* does not hold in practice. In such a case, it is questionable if an abstract workflow should be used in favor of simply using a concrete workflow. We tried to cope with this problem by supporting a mixture of abstract and concrete tasks, so that workflows containing only concrete tasks are also supported. Second, the usage of our approach might lead to a unpredictable test generation and execution. This will be an issue in cases where an automatic task mapping strategy, i.e. service-based task mapping, is used and the mapping to concrete tasks may change in every execution of the workflow[3]. While this can be seen as a benefit, because we can introduce new test algorithms easily, it may be an issue when strong reproducibility is required. We think that our approach is still applicable in such cases by using either a manual task mapping, or when using our prototypical implementation, use OSGi filters which match to one and only one service instance. Another way to ensure reproducibility is to apply *data provenance* (Deelman et al., 2009) by recording the (complete) history of the creation of data object (in our case elements in the MBT model) to be able to reproduce the results of the workflow execution. Another approach would be to go into the details of formalizing the execution semantics of the concrete tasks. In our formalization we avoided this step, because we think that no formal language for describing the execution semantics of tasks has become widely adopted. In order to keep our approach as general as possible, we only formalize the artifacts of MBT in a metamodel and let the concrete tasks operate on an instance of this metamodel. This way, the concrete tasks can be implemented in any formal or general purpose programming language.

Additionally to the above issues, the question about the *costs* to create and maintain the refinement of the MBT metamodel and the abstract workflows may arise. Within this dissertation, we did not asses this issue with empirical studies, but left it for future work as the next logical step after this dissertation.

[3] this is analog to a polymorphic dispatch in object-oriented systems, where the object determines the behavior of a method call

3.7. Related Work

Metamodels have been created for many domains, like the *Common Warehouse Metamodel* for data warehouse integration (Poole and Mellor, 2001), the *AUTOSAR* standard (Heinecke et al., 2004) for automotive software architecture, or to model *Service Oriented Architectures* (Friedenthal et al., 2008). In the domain of testing, the UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts of test systems (Object Management Group, 2005a). This language is orthogonal and supplementing to our work. In our work, we created a metamodel for the test generation methodology MBT. The output of this approach is a test suite (a set of test cases), which describe the expected behavior of the SUT. The UML Testing Profile has a much broader design aim and our test suite can be one artifact in a model of the test system described by the profile.

Several modeling notation have been used to generate test cases using the MBT approach, i.e. UML Interaction Diagrams (Nayak and Samanta, 2009) and UML Use Case Diagrams (Basanieri and Bertolino, 2000). Compared to our work, we proposed a MBT metamodel which is independent of the modeling language, so we think that any of these modeling notations can be used.

The workflows (or similar concepts where a defined order of task is specified) of MBT has been identified by previous work. For example, (Utting et al., 2012) identify *the process of MBT* on a very high level, and (Calvagna and Gargantini, 2010) used a *Generation process of a combinatorial test suite* showing typical concepts of workflows. However, to our best knowledge, our work is the first attempt to formalize the workflows of MBT using abstract workflows. The concept of abstract workflows have been used in different workflow systems, like VisTrails (Callahan et al., 2006), Taverna (Oinn et al., 2004), and P-GRADE (Kertész et al., 2007). These systems support different task mapping strategies, like user-defined mapping, mapping with an internal scheduler, or mapping with an external broker. Our work relies on these concepts by using abstract workflows to model the test generation workflow of MBT.

3.8. Conclusion

In this chapter we coped with the problem of formalizing the MBT approach with the aim of automating the test generation. To come up with a formalization for that purpose, we identified that MBT has two aspects to formalize, namely the *artifacts* and the *workflows* of MBT. For both aspects, we presented the criteria we used to choose a fitting formalization language. We chose *metamodels* and *abstract workflows* to formalize the artifacts and the workflows of MBT. A MBT metamodel was proposed and the integrated concepts were described. Later, it was shown how abstract workflows together with a *task mapping strategy* can be used to model the test generation workflow. A prototypical implementation, based on the Eclipse Platform, was presented to show the practicability of our approach. We used the Eclipse Modeling Framework (EMF) to create the MBT metamodel, and extended the Modeling Workflow Engine (MWE) to support abstract workflows and service-based task mapping. We concluded the chapter by discussing the advantages and implication of our approach, and presenting the related work.

CHAPTER 4

Model-based Testing Of Component Systems

4.1. Introduction

In the last chapter we presented our approach to formalize MBT with the goal of automating the test generation. Our approach uses metamodels to formalize the artifacts of MBT and abstract workflows to formalize the processes of MBT. We presented the results of our approach, namely the common MBT metamodel and the idea of using abstract workflows to support families of MBT approaches, independent of any target system or modeling notation. In this chapter, we present a refinement of our approach. We will formalize MBT for component systems using the UML notation. The results of this chapter will serve as an intermediate step to show the hypothesis of this dissertation. We will give an answer to the question wether MBT can be used for systematic and automated test case generation and execution for component systems. The answer to this question is already known to the scientific community because MBT has been successfully applied to component systems. However, we think that presenting this intermediate result is important for this dissertation. It later allow us to show that MBT of DCSs can be achieved by *applying* the formalization presented in this chapter, rather than creating another layer of refinement.

The goals we aim to reach in this chapter are the following:

- Create a formalization of MBT for component systems using the UML modeling notation, and preserve the original goal to create a formalization that allows to automate the test generation. Thus, the formalization should be detailed enough to allow the creation of UML-based test models, some test selection criteria, and the execution of

the test generation workflow.

- Show that our approach to formalize MBT with metamodels and abstract workflows is applicable to one target system type and one modeling notation by implementing the results of this chapter in a software prototype.

To reach these goals following steps are taken in this chapter:

- We place our MBT refinement in a MBT taxonomy, including the target system type, to be able to create a precise formalization (Sec. 4.2).
- On the basis of the classification of the target system type, we show how the UML can be used to create test models. We use UML class diagrams and UML Statecharts (Sec. 4.3).
- We present a formalization for test suites fitting the UML-based test models. The most important part will be the inputs (stimuli) to the SUT and the outputs from the SUT (Sec. 4.4).
- We show how test selection criteria and test case specifications can be modeled. We focus mainly on structure-based test selection criteria but will also present an approach to specify explicit test case specification using temporal logics. We further show how model-transformations can be used to express the semantics of test selection criteria (Sec. 4.5).
- We formalize the semantics of the test case generation with *model checkers* by defining model-transformations of UML-based test models to a model checking problem, of test case specifications to temporal formulas, and the interpretation of counterexamples as test cases. (Sec. 4.6)

4.2. Classification Of The MBT Refinement

Before going into the details of the formalization, we first characterize the target system type, for which we will refine the MBT formalization. The application example of this dissertation is embedded in a scenario of wirelessly communicating nodes, which cooperatively execute a typically long-running task, such as sensing seismologic activity in order to throw early warning alarms for an earthquake (Fischer et al., 2012). In this scenario, we focus on the component-based software of *a single node*. We see each individual component on this node as a *reactive*, *discrete*, *untimed* system, so we exclude *transitional*, *continuous*, and *timed* systems. Component interaction is done using an *event-based* or *method-based* com-

munication paradigm, both *synchronous* or *asynchronous*. We therefore exclude *stream-based* communication. Each component instance is expected to consume one event or method call at a time. Processing a received event or a method call will therefore be *deterministic* (not interrupted by other events of method calls). However, we assume that the component model specifies that component instances run in parallel. When seeing the component system as a whole, the scheduling of component instances may therefore introduce *non-determinism*.

We further classify our MBT refinement using the taxonomy of Utting et al. (2012) (Fig. 4.1). This taxonomy defines seven dimensions for the various concepts and approaches to MBT. In our MBT refinement, the subject of our test models is mainly the *SUT*. The test models are assumed to be *separate* from development models. The characteristics of our test models reflect the characteristics of our target system type, as defined above. The modeling paradigm of the test models is *transition-based*, as we will use UML Statecharts. Our test generation method is driven by *structural* test selection criteria and *test case specifications* expressed in temporal logics. Finally, the test generation technology is based on *model-checking*, and we generate tests *offline*.

The used taxonomy is limited in not dealing the *degree of knowledge* about the structural and behavioral details of the SUT. If the test model recreates the structure and behavior of the SUT in detail, we call the approach *white-box*. However, if the test model abstracts over the concrete structure or behavior of the SUT, we call the approach *black-box*. Some MBT approaches combine black- and white-box into a so called *grey-box* approach, where certain parts of the SUT are known in more detail than others. Our MBT approach can be classified as *black-box* because we assume a fair amount of abstraction from the details of the SUT by focusing on stimuli (inputs) and expected behavior (output).

4.3. Test Models

The test model is the central artifact in the MBT approach. It is typically created manually by a test engineer using some requirements specification of the SUT. To be able to create the test model, the test engineer needs a modeling language. This modeling language should fit the target system type of the SUT. In our approach, we use the Unified Modeling Language (UML) specified by the OMG (Object Management Group, 2005b). The UML is a widely accepted language in the software industry, and many vendors provide tools with support for graphical modeling, analysis, simulation, and code generation. In the version 2.1, the UML defines 13 diagrams and graphical notations for describing the structural and

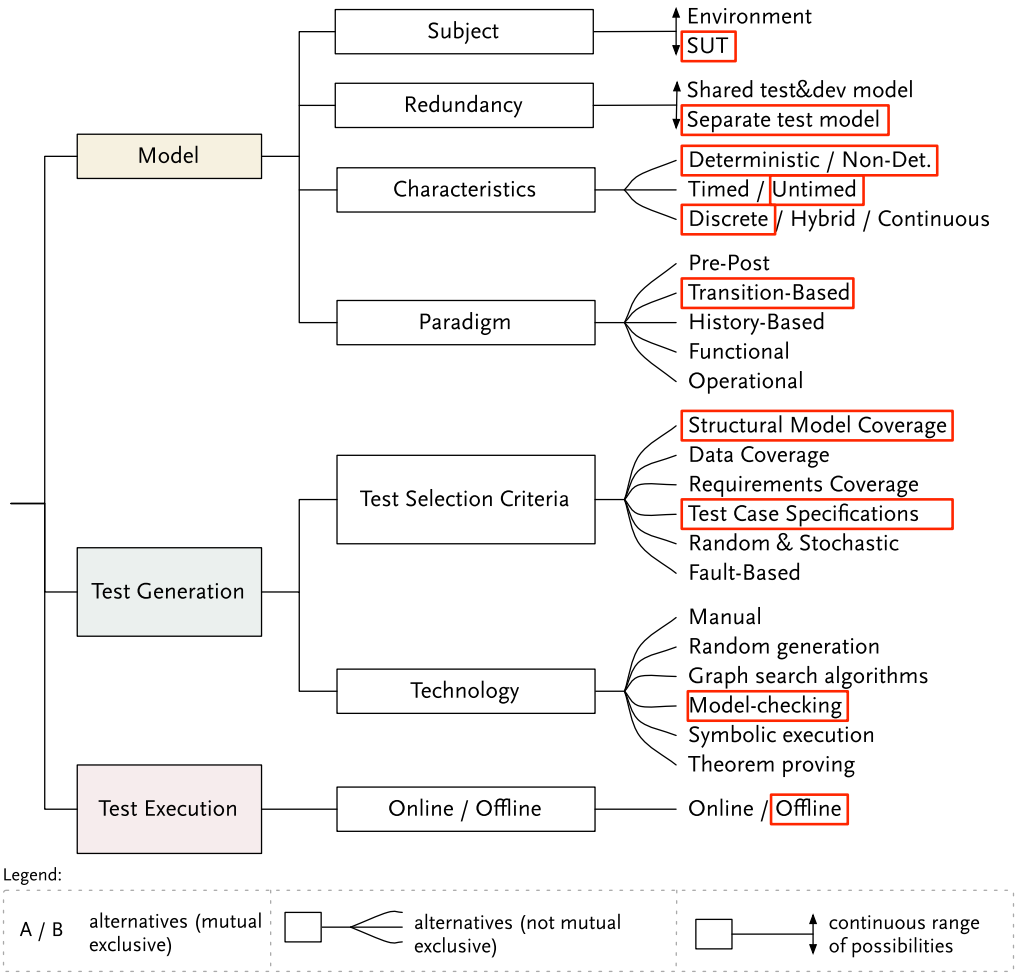


Figure 4.1.: Classification of the MBT refinement using the taxonomy of (Utting et al., 2012). The red boxes mark the selection in each dimension of the taxonomy.

the behavioral properties of systems. Out of this rich set of notations, we decided to use the *UML class diagrams* and the *UML Statecharts* to create test models. The UML is a semi-formal general purpose language for different target system types, and thus provides some inherent modeling freedom. For example, it is the decision of the modeler which diagram and notation she uses for describing the behavior of a system. In addition, the UML has several semantic variation points, which makes it necessary to explicitly define semantics when ambiguities have to be avoided. As we will see in this section, one of the main variations allowed in UML Statecharts are the expression language used for *guards* and *actions*. To avoid ambiguities, we present in this section the sub-set of the UML we considered for our approach.

We introduce our concepts using an example application called *Online Storage*: The Online Storage offers users a service to store arbitrary files for sharing or backup purposes. Users of that service are provided with a *desktop client*, with which they can upload their files. These files are received and stored by a *server*, which also handles the authentication of the users. Because the service is quite new, the service provider decided to give every user a free contingent for uploads. The Online Storage service has not existed for long, so the set of features is limited. The requirements of the service has been written in terms of *user stories*, a lightweight specification approach used in many agile software development methods. The following lists contains some of the requirements:

- *As a user, I want to upload a file using the client so I can backup my files.*
- *As a user, I want to send up to 10 files to the server for free so I can check out the service.*
- *As a user, I want to cancel the transmission of a file at any time so I can avoid decreasing my contingent by uploading a wrong file.*
- *As a client, I want the request for a file transmission to be approved so only registered users can use the service.*
- *As a client, I want to be informed when the upload succeeds so I can decrease the contingent of the user.*
- *As a client, I want to be informed when an upload is aborted (due to cancellation or the server running out of storage space) so I can give feedback to the user through my user interface.*
- *As a server, I can only handle one client and its transmission at a time.*
- *As a server, I need to abort any active file transmission if my hard disk is full so I can avoid losing user data.*

These requirements can be used to develop the client and its user interface, the server, and their interaction. However, we assume that these requirements are also used to create a separate specification of the expected behavior in terms of a test model. Recalling that we restricted our MBT refinement to black-box testing, the structure of this test model can be different from the structure of the implementation, and some behavior can be excluded. We applied following abstractions to the Online Storage example, which an implementation normally has to deal with:

- network distribution of client and server
- network aspects like protocols and timeouts during the file transfer
- details of the graphical user interface like window-management
- aspects of user authentication mechanism like key-management
- detection of full disk

Using the *Online Storage* example, we now can go into the details of our modeling concepts. Test models contain *structural* and *behavioral* parts. The structure contains entities, their hierarchy, attributes, and relation between entities. The behavior specifies the stimuli and the reaction of the system in different situations.

Structure

In our approach, we describe the structure of test models using UML class diagrams, and use *UML components* and *UML interfaces* within these diagrams. An UML component encapsulates *properties*, *methods*, and *signals*, and can be *passive* or *active*. Properties can be *attributes* or *references*. Attributes are typed as Boolean, Integer, or Enumeration with an initial value. For attributes typed as Integer, a *value range* must be specified. The grammar for the value range language is presented in Appendix A. Properties can also be *references*. References are used to model the potential interaction between UML component instances. References have a cardinality, which describes the potential number of objects which can be stored in the reference, and if any object is mandatory. Possible values are:

- $0..1$: Optional and unary
- $0..n$ (or just n): Optional and multiple
- $1..1$: Mandatory and unary (Default)
- $1..n$: Mandatory and multiple
- where n is a natural number greater than one or $*$ (denoting unbounded multiplicity).

References are used to model signal-based or method-based communication. The receivable signals or callable methods are determined by the target of the reference, which can be a UML component or UML interface declaring signals or methods (more details on communication behavior will be shown later). Finally, a UML component can be modeled as *active* (shown as class box with an additional vertical bars) or *passive*. Active UML component instances start to execute their classifier behavior as a direct consequence of their creation, and do not stop until either the behavior is finished or the object is terminated by some external object (Object Management Group, 2005b). In our approach, the classifier behavior of UML components is described using UML Statecharts (see next section).

Test models can be treated as normal artifacts in a software lifecycle, and so manageability and reusability are issues with them. Our approach to support these issues is to allow the concepts of *interfaces* in test models. UML interfaces are an optional means of design to make test models reusable by allowing UML components to inherit their structure. A vital example for using an interface is to model a specific role of a component. This reduces the coupling between the user of the component and the component, because only the necessary attributes, methods, and signals are visible to the user. Like UML components, UML interfaces encapsulate *properties*, *methods*, and *signals*.

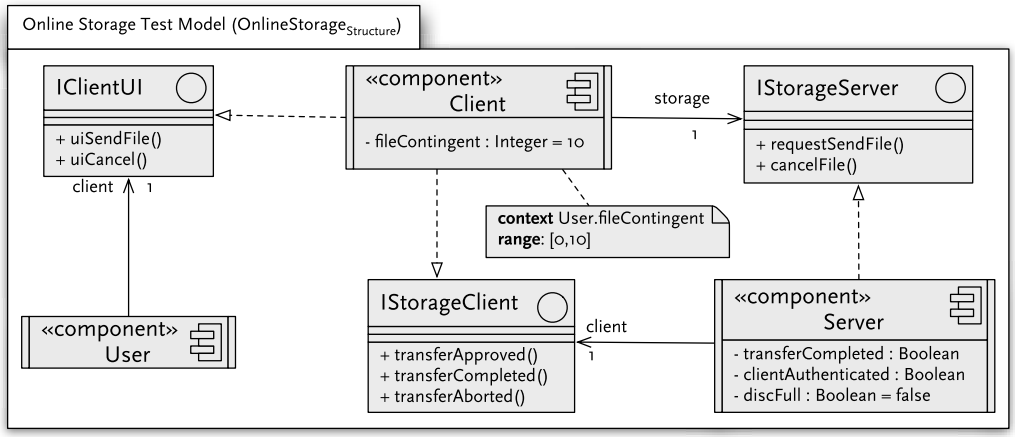


Figure 4.2.: Test model structure for the Online Storage example using UML components and UML Interfaces.

Figure 4.2 shows the structural part of our test model for the Online Storage example. Based on the requirements, we decided to create three UML components named User (representing the user of the Online Storage), Client (representing the desktop application), and

Server (representing the file storage server). We also created three UML interfaces to make the roles of the components clear. For example, the Client has the role of the user interface by implementing the UML interface IClientUI which the User leverages on. In addition, the Client and the Server communicate though the roles of the IStorageClient and IStorage-Server. The test model also contains several attributes. For example, the file contingent of the user is modeled using the fileContingent attribute of the Client. The value range of this attribute is further limited to positive integers (including zero) up to 10. Its default value is set to 10 according to the requirements. Other attributes like transferCompleted, clientAuthenticated, and discFull reflect our mentioned abstractions of the implementation, and represent potentially complicated facts as boolean switches. Finally, the potential interaction between the three components is modeled using references and a signal-based communication paradigm. Cardinalities are kept simple in this example.

Behavior

The second aspect of test models is the capability to describe the expected behavior of the SUT. In our approach, we use UML Statecharts for that purpose. Statecharts were first introduced by Harel (Harel, 1987) as a visual formalism for modeling the behavior of reactive systems. Harel describes Statecharts as *finite state machines* (Gill, 1962) extended with *hierarchy*, *parallelism*, and *communication*. The common Statecharts semantic is that the described state machine is always in one of a finite set of *active* states. When an event occurs, the system reacts with an action, such as changing the value of a variable or taking a *transition* to another state [1]. An exemplary UML Statechart is shown in Fig. 4.3.

States

The state S0 is an *initial* state and denotes the first active state. The top-level state S1 contains an initial state S2 and one other state S3. State S3 is a *orthogonal* state (originally called *AND* state by Harel) and is devided by a dotted line in two regions. These regions run in parallel, so that whenever S3 is active, both contained regions are also active. The upper region of S3 is a *combined* state (originally called an *OR* state by Harel) containing the states S4, S5, and S6. Whenever a combined state is active, one of its contained states is active *exclusively*. The lower region of S3 contains a *choice* state S9, where branching of the control flow can happen.

[1] For a detailed presentation we refer to (Harel and Politi, 1998) and (Utting and Legeard, 2006).

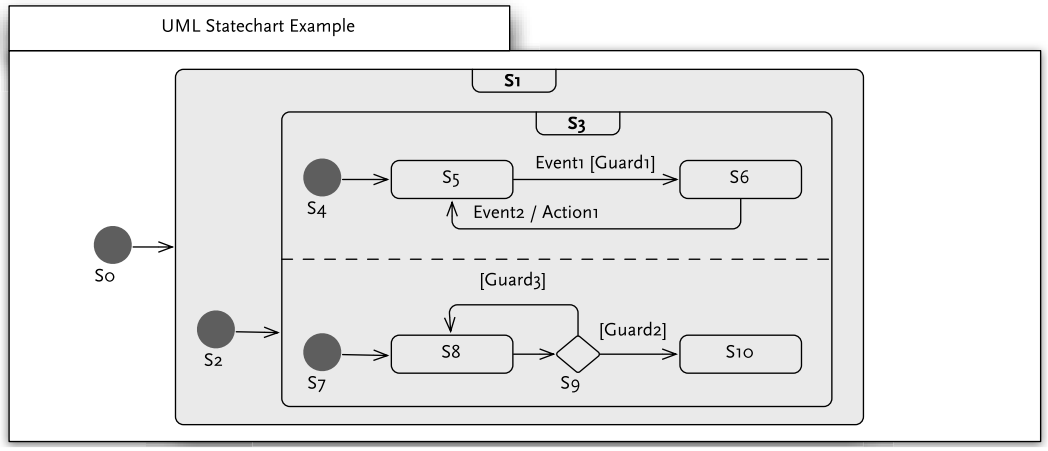


Figure 4.3.: An exemplary UML Statechart.

Transitions

Transitions are used to change the set of active states. They are labeled with *triggers*, *guards*, and *actions* with the following syntax: $Trigger[Guard]/Action$. All parts of the transition label are optional. $Trigger$ is the name of the event (signal reception or method call) to be processed. $Guard$ is a boolean condition that must evaluate to true to make the transition happen. The $Action$ label describes the effects of the transition. An $Action$ can update variables and produce new events. A transition becomes active if the source state of the transition is active, the fitting event is triggered, and its guard evaluates to true. In this case, the source state of the transition becomes inactive. Then any action of the transition is executed and the target state of the transition is activated. Atomicity of transitions is maintained using the *Run-To-Completion* (RTC) semantics (Object Management Group, 2005b), which require that an event can only be processed if processing of the previous event has been completed.

Semantic Variations

Beside these explicit semantics, the specification for UML Statecharts contains some semantic variation points, and many attempts to formalize the semantics have been proposed. Two important aspects of the semantics are the languages used for guards and actions. For both aspects, we created a self-defined language, which is inspired by the *Object Constraint Language* (OCL) (Object Management Group, 2006b) and Java. The complete grammar for

the guard and action language is presented in Appendix A. In a nutshell, guard actions are boolean valued expression containing typical logical and arithmetical operations. Like in OCL, guard actions can access structural elements (like variables, methods, and signals) of the context UML component. Action expressions can be a sequence of assignments to variables of the context UML component, or raising signals or method calls. Action expressions therefore allow communication between UML Statecharts. To maintain the RTC semantics in a scenario of asynchronous communicating components, the consumption of received events is serialized using a First In – First Out (FIFO) input buffer for every UML Statechart. However, the input buffer is not modeled explicitly in the test models. Instead, as we will show later, the addition of the buffer is controlled by configuration parameters of the test generation workflow.

Non-Determinism

Our last concept for the creation of test models is the allowance of *non-determinism*. There are four sources of non-determinism in test models based on our approach: First, each state machine instance (created as a direct consequence of the instantiation of the active context UML component) runs concurrently to other state machine instances. This concurrency implies that some kind of *scheduling* strategy has to be chosen, which introduces an inherent non-determinism to a test model. Second, a test model where several transitions may be active at the same time causes non-determinism. In this case, a transition is chosen randomly. Third, the usage of orthogonal states, which contain two or more parallel running regions, causes non-determinism. Finally, we introduce the concept of *Free Attributes*:

Definition 4.1 (Free Attribute). A *free attribute* is an attribute of an UML component which

- has no initial value,
- and is not modified by the action expressions of any transition of the behavior describing Statechart.

Free attributes cause non-determinism, because the value of such an attribute can be assigned with an arbitrary value by the environment, respecting potential range restrictions, at any time.

Coming back to our Online Storage example, figure 4.4 shows the behavioral part of our test model. The UML components User, Client, and Server where modeled as active components, so we specified an UML Statechart for each of them. The Statechart of the User component describes the expected behavior of the user of the Online Storage, and therefore

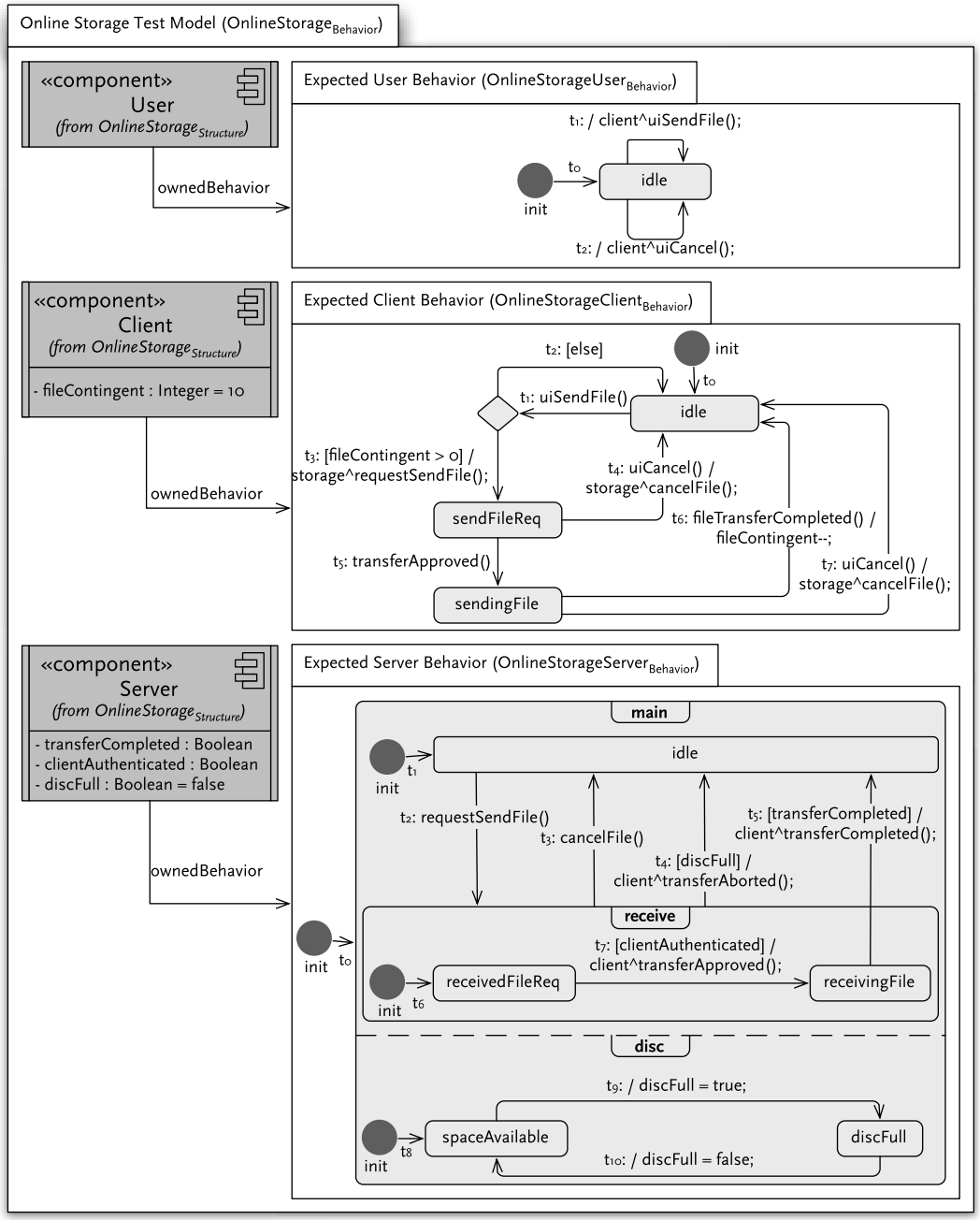


Figure 4.4.: Test model behavior for the Online Storage example. For each of the active UML components, the behavior is specified using a UML Statechart. These UML Statecharts run in parallel and communicate signal- or method-based.

represents a part of the environment's behavior. In this example, we modeled the behavior of a user in such a way that she can send a file and cancel a transmission at any time. Since both outgoing transitions of the state idle have no trigger and no guard, their selection is randomly, which adds non-determinism to the test model. The expected behavior of the UML component Client is the reaction to user input, the management of the file contingent, and the communication with the server. The UML component Server manages the communication with the client, the file transfer, the authentication of the user, and the handling of a full disc. The abstractions we made add additional non-determinism to the test model: First, we use an orthogonal state for separating the main functionality of the server from the full disc recognition. Second, we modeled the properties transferCompleted and clientAuthenticated as free properties, which means that these properties can are changed by the environment to signalize the completion of the file transfer, or the validity of the user authentication.

MBT formalization refinement

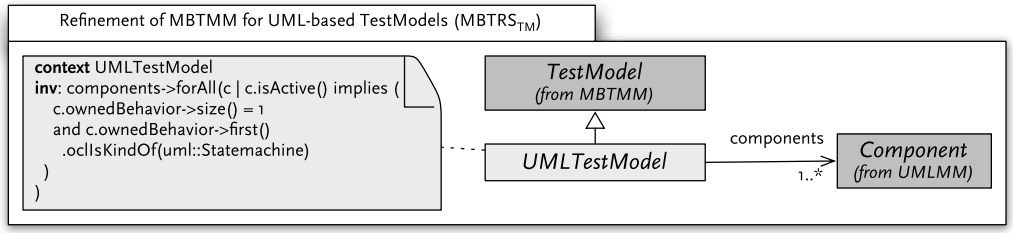


Figure 4.5.: Refinement of the TestModel concept for UML. UMLTestModel refers to UML Components. If a component is marked as *active*, it must have exactly one Statemachine describing its behavior. This figure refines elements from Fig. 3.1.

To complete the formalization of our test model refinement for component systems using UML, we present the refinement of the common MBT metamodel (Sec. 3.2) in figure 4.5. The element UMLTestModel inherits from the abstract TestModel element. UMLTestModel references a collection of passive or active UML components. Together, they represent the structure of the test model. An additional OCL constraint specifies that an *active* UML component needs to have exactly one UML Statechart, representing the expected behavior of that component.

4.4. Test Suites

In the last section we presented the refinement of the TestModel concept of the MBT metamodel for reactive component systems using the UML. In this section, we present the corresponding refinement for the TestSuite and TestCase concepts. Recalling that a test case represents a subset of the overall behavior defined by the test model, our test cases represent a subset of the behavior of our UML based test models. The behavior of these test models is described by several, parallel executed automatons which communicate using a method based or signal based communication paradigm. To create a formal definition for test suites and test cases using metamodels, we first want to introduce the involved design decisions:

Self-Contained and Reliant Tests

Test cases can be designed to include sufficient information so that the original test model is not needed. On the other side, they also can be designed to include as little as possible information, but requiring the original test model. In the first approach, the test cases are *self-contained*, which means that they include all behavior necessary to execute them without the need of the original test model. This kind of test cases are handy if we want to avoid giving our test models to other parties. However, these test cases usually need more disc (or memory) space, and changes to the test model may imply the regeneration of the whole test suite. In the second approach, the test case contains minimal behavior, but is *reliant* on the original test model. The expected behavior of the SUT has to be reconstructed by using the original test model. This kind of test cases consume less disc (or memory) space, and allow changes (to some degree) to the test model without the need to regeneration. However, it also forces us to reveal our test models to other parties.

Non-Determinism

Another design decision is to which extent test cases should handle non-deterministic choices of the SUT. If non-deterministic choices of the SUT are not handled, we may recognize a difference between the output of the SUT and the expected output of the test case, although the SUT is correct. A test case might falsely detect a fault, so an inconclusive outcome would be the correct verdict in this case (Fraser, 2007). One design approach is to expect specific decisions of the SUT, i.e. scheduling of components, and fail if the expectations are not fulfilled. Another approach is to annotate places in the test case where non-deterministic

behavior is possible to avoid a falsely detecting a fault. Finally, test cases can be designed to have a tree-like structure, so that alternative behavior can be specified for branching points.

Attribute Values

Our UML based test models can contain attributes in the UML Class diagrams. A design decision is to whether store *concrete* or *symbolic* values for attributes in the test cases. A *concrete* value means that we expect a single correct outcome of the SUT for this attribute. In contrast, a *symbolic* value defines a set of possible value outcomes. We can use symbolic values in two ways: First, we can use them to derive several test cases containing concrete values. The concrete values could be chosen by enumerating the set, if possible, or by heuristics like choosing edge values (Utting and Legeard, 2006). Second, we can use symbolic values to allow non-determinism in the SUT (see above).

Concrete values improve the self-containment of test cases, so that consumers of these test cases do not need to derive concrete values in order to be able to execute them. Symbolic values allow to derive more than one test case by selecting concrete values for the symbols. However, with symbolic values, the test cases are not self-contained anymore, and the consumer of the test cases has to perform additional steps to derive executable test cases.

Supporting all possible combinations of the above design decisions in one formalization would lead us to a complex metamodel, crowded with unnecessary details for some use cases. The formalization of reliant test cases, for example, does need less concepts than one for self-contained tests, since the missing information can be derived using the original test model. We therefore decided to create a basic formalization for reliant test cases, and show step by step how extensions to this basic formalization can lead us to self-contained test cases. In addition, we will show an extension for non-deterministic test cases.

The basic refinement of the MBT metamodel for UML is shown in Fig. 4.6. The most interesting elements are the UMLInput and UMLOutput, with their specializations for method and signal based inputs (stimuli) and outputs (observations). In addition, every UMLTestStep references an scheduled UML component through the relation scheduledComponent. With this basic refinement, it is possible to express reliant test cases, which only contain a series of inputs and outputs.

Figure 4.7 shows a reliant test case for the Online Storage Example using a UML sequence chart. This test case contains only signals between the involved components (User,

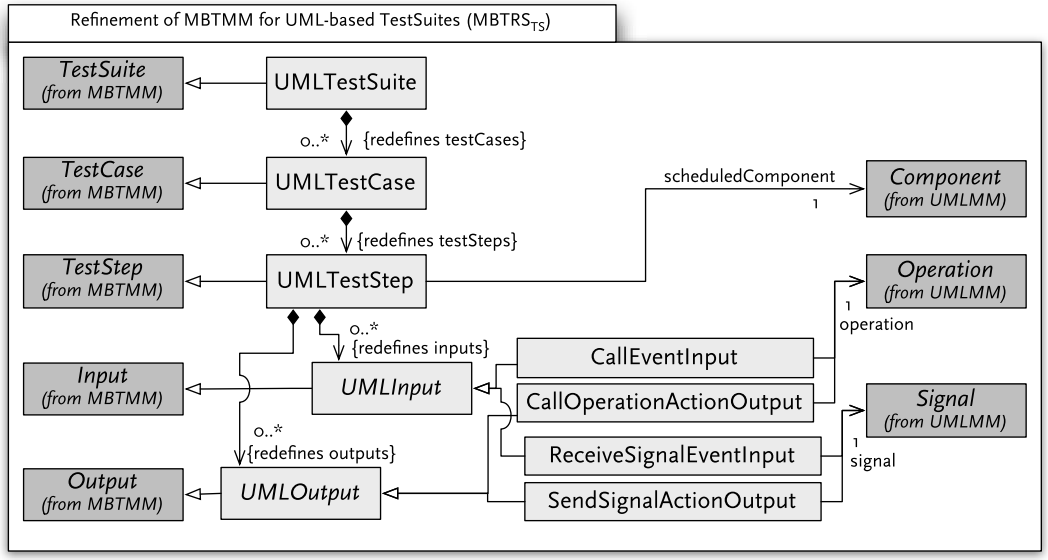


Figure 4.6.: Refinement of the TestSuite concept for UML. The main refinements are for TestStep, Input, and Output, with each of them referencing the appropriate elements of the UML metamodel. This figure refines elements from Fig. 3.1.

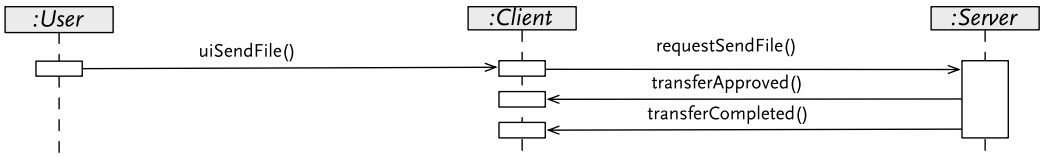


Figure 4.7.: Basic reliant test case for the Online Storage Example containing only signal inputs/outputs. In this test case, the user uses the client to send a file to the server, the server authenticates the user, the client starts the transfer, and finally the server notifies the success of the transfer.

Client, and Server), but still sufficient information to be executed against the SUT. To enhance the test case with more details, we can use the original test model (Fig. 4.2 and Fig. 4.4), and simulate the state machine of the test model while executing the test case against the SUT. We then can derive more expectations about the behavior of the SUT using the state of the simulation. This technique has been used successfully in existing MBT approaches.

If we need to reduce the dependencies of the test cases to the original test model, we have to extend the basic refinement to include more details. Our first extension is to include the expected value assignments of attributes in test cases. Figure 4.8 shows the resulting metamodel.

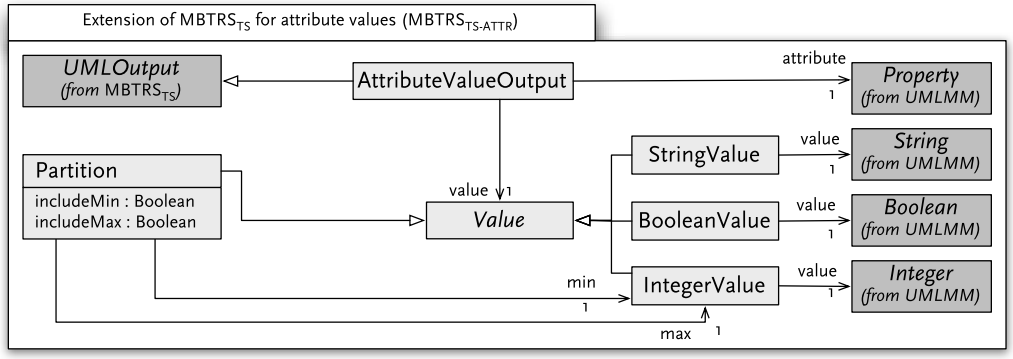


Figure 4.8.: Extension to the basic refinement of the MBT metamodel for UML with expected value assignments of attributes. This figure extends elements from $MBTRS_{TS}$ metamodel (Fig. 4.6).

In this metamodel, the element AttributeValueOutput models the expected value of an UML attribute in a test step by extending the UMLOutput element from the basic formalization for reliant test cases. The value of the UML attribute is modeled using a type system including String, Boolean, and Integer types, with each of them referring to the respective UML type. With this addition, we can enhance our previously reliant test case for the Online Storage example with *concrete* attribute values (Fig. 4.9). To support *symbolic* values of attributes in test cases, we added the Partition element to our metamodel. This elements allows us to declare a value range (min, max) for possible correct outputs of the SUT.

The next extension to the basic formalization is to include structural information of the Statechart in test cases. Depending on the domain or application scenario, we might have a SUT which is implemented using some automaton concept. For example, the automotive

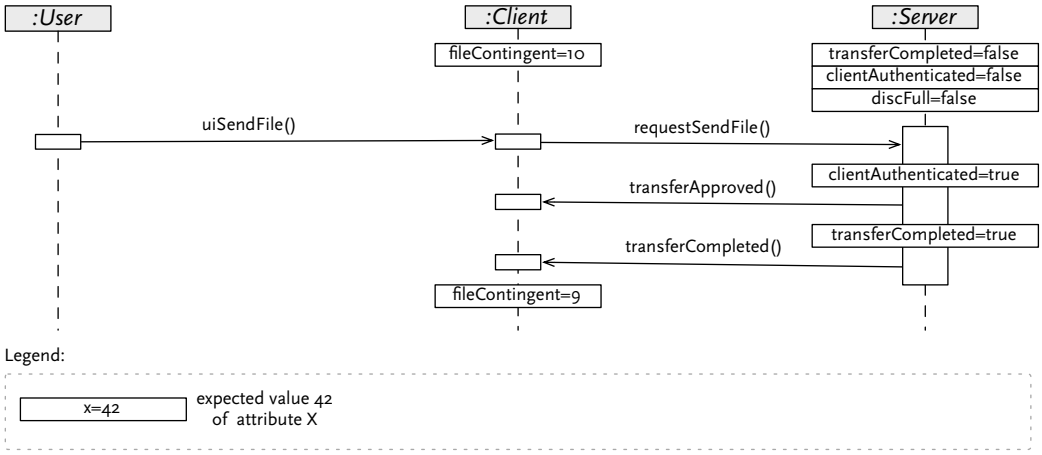


Figure 4.9.: A test case for the Online Storage Example containing signal inputs/outputs and attribute values. In contrast to the reliant version (Fig. 4.7), the original test model is not required for deriving attribute values during test execution.

industry makes intense use of automata8 for design, verification, and implementation of car components. The automaton is typically transformed into programming code, where the information of states and transitions might get lost. However, if the system preserves the information of states and transitions during runtime, we might interpret the current active states and transition as another *output* of the system. A reasonable way to test such a system using MBT is to include the information about states and transitions in test models. Our approach to use UML Statecharts for test models allows testing such systems. With reliant test cases, we can derive the expected active states and transitions of the SUT during test execution. However, for a self-containing test, we need to include the structural information in our test cases. Fig. 4.10 shows our extension to the basic formalization of reliant test cases. We created two sub-types of the element UMLOutput for the configuration (def. 4.4) and active transitions. Both of them reference the respective UML types for states and transitions. With this addition, we can enhance our test case for the Online Storage example with structural information (Fig. 4.11). Test cases containing details using all the above extensions (signal input/output, attribute values and structural information) can finally avoid using the original test model, since all information is contained in the test case itself.

To broaden the view on the various presented metamodels, figure 4.12 shows an overview of the current state of our MBT refinement. We extended parts of the common MBT meta-

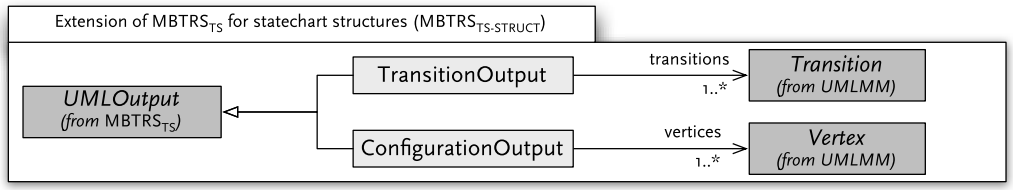


Figure 4.10.: Extension to the basic refinement of the MBT metamodel for UML with expected transition and configuration outputs. This figure extends elements from $MBTRS_{TS}$ metamodel (Fig. 4.6).

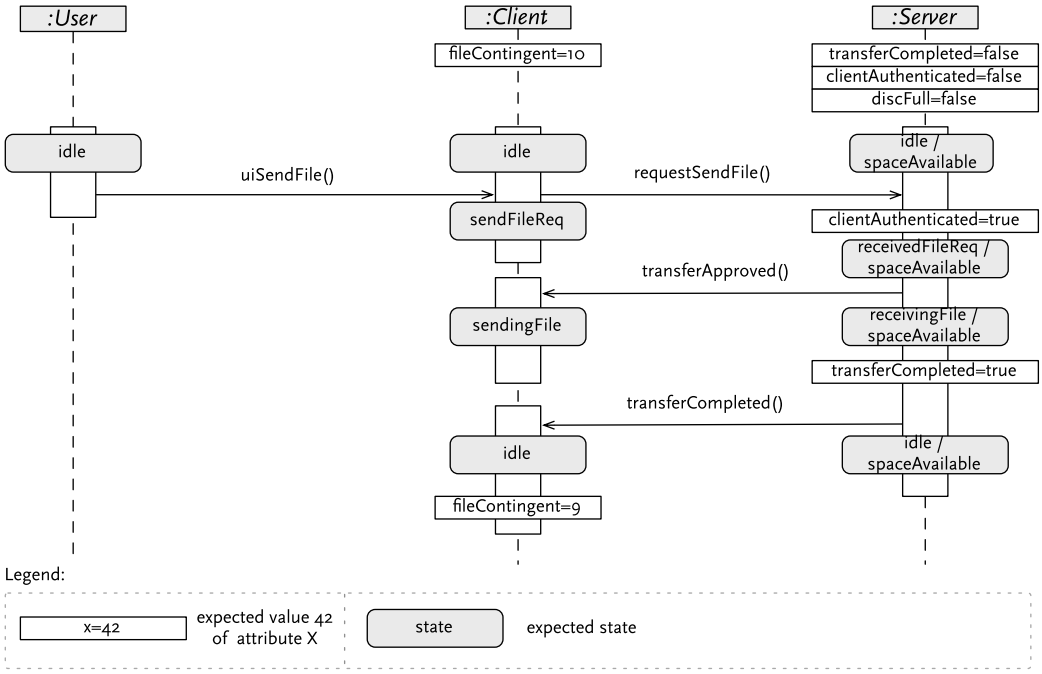


Figure 4.11.: We omit the transitions from the figure for readability purposes.

model $MBTMM$ with UML Statechart specific concepts. The metamodel $MBTRS_{TM}$ refines the basic concept of test models by referencing a list of UML components, with each of them having a Statechart describing the component's expected behavior. Basic test suites for UML Statecharts are formalized with the $MBTRS_{TS}$ metamodel. This metamodel defines basic communication concepts for sending and receiving signals and calling operations. Using a stepwise extension of this basic metamodel, we were able to express the expected values of attributes by either concrete or symbolic values in the $MBTRS_{TS-ATTR}$ metamodel. Finally, with the $MBTRS_{TS-STRUCT}$ metamodel we allowed test cases to contain structural information for the configuration and active transitions.

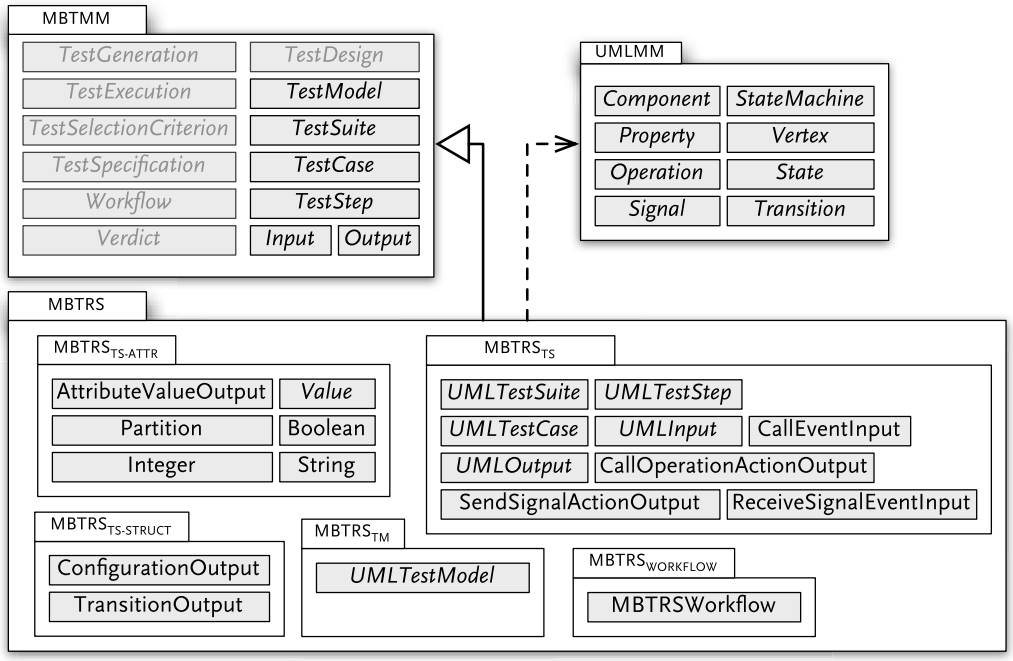


Figure 4.12.: Overview of the refinement of the common MBT metamodel for reactive systems using UML made in this section. We modularized the *MBTRS* metamodel into five packages, so that we can choose between the level of details in test cases.

4.5. Test Selection

In this section we show how test cases can be selected from our UML based test models. We refine the *test selection criteria* and the *test case specification* concepts aligned with our previous formalization of UML based test models and test cases. We present our approach using five representative test selection criteria, and show how model-to-model transformations can be used to formalize the relation between test selection criteria and test case specifications. We decide to focus on *structural* test selection criteria (Utting and Legeard, 2006; Binder, 1999; Harel and Politi, 1998), which are obvious candidates when using UML Statecharts as the modeling notation for test models. They are derived from key concepts of the modeling notation like states and transitions and are used to deal with the coverage of the control-flow of the test model. In addition, we show how *explicit test cases* can be expressed to give precise control over the generated tests.

4.5.1. All-States

Definition 4.2 (all-states). All-States coverage is achieved if every state of the model is visited at least once.

Applying the definition of all-states (Def. 4.2) to our test models, full coverage is achieved when our test suite visits every vertex (simple state, pseudo state, combined state) of the UML Statechart at least once. For example, in our Online Storage Example (Fig. 4.4) the sequence of transitions $t_0; t_1; t_3; t_5$ gives *all-states* coverage for the Client component. This corresponds to following actions in the SUT:

- client is ready (t_0),
- the user uploads a file using the UI (t_1),
- the client (while having a sufficient file contingent) requests sending the file to the storage (t_3),
- and the storage approves the file transfer (t_5)

We model all-states as an refinement of the common metamodel (Fig. 3.1) in Fig. 4.13. We introduce the element UMLTestSelectionCriterion, which extends the TestSelectionCriterion element of the common metamodel. UMLTestSelectionCriterion is then extended by the element All-States representing an intensional description of a sub-set of the test model behavior. All-States redefines the target of the reference describes to a new element VertexTestCaseSpecification. VertexTestCaseSpecification represents an extensional, model-specific test

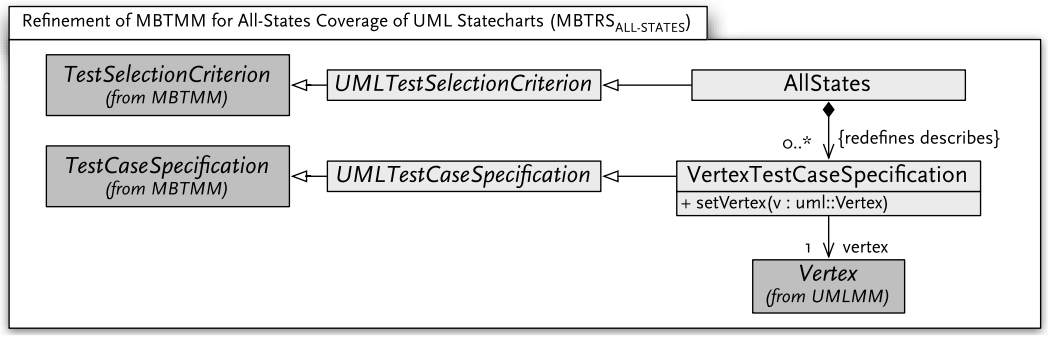


Figure 4.13.: Refinement of the common metamodel for the UML Statechart specific test selection criterion *all-states*. This figure refines elements from Fig. 3.1.

case specification. It references a single, mandatory Vertex element from the UML metamodel – the state to cover.

Having the structural refinement of our concepts, we need to find a way to describe the semantics of test selection criteria. The semantics are simply the answer to the question how the intensional description of the criterion relates to the extensional. Since both the source (element All-States) and the target (element VertexTestCaseSpecification) are defined using a metamodel, we can use model-to-model transformations to describe their relation. The model transformation takes the $MBTRS_{ALL-STATES}$ both as input and output, and thus is an *in-place* transformation (Czarnecki and Helsen, 2006). For describing the transformation, we use the textual notation *Xtend* (Eclipse, 2012). However, our approach is not tied to this language, and we could use other model-to-model transformations like QVT (OMG, 2011) as well. We shortly introduce the Xtend syntax, but refer the reader to the comprehensive documentation for more details (Eclipse, 2012).

Xtend is a model-to-model transformation language with a textual notation. It allows traversing and modifying metamodels using an textual expression language with a syntactical mixture of Java and OCL. A Xtend transformation consists of functions which operate on metamodel instances. Functions can take metamodel elements as parameters. The transformation for all-states in listing 4.1 shows two functions. The first function toTestCaseSpecification(AllStates,TestDesign) is provided with AllStates and TestDesign elements, while the second function is provided with a Vertex element. Xtend allows traversing metamodels through accessing properties and references using a Java-like dot-notation. For example, we access the describes reference of the AllState element using allStates.describes in our transformation. Further than traversing, Xtend supports the modification of metamodels

```
1 toTestCaseSpecification(mbtrs::AllStates allStates, mbtmm::TestDesign td):
2   allStates.describes.addAll(
3     td.testModelComponents().vertices()
4       .toTestCaseSpecification()
5   );
6
7 create VertexTestCaseSpecification toTestCaseSpecification(uml::Vertex v):
8   setVertex(v);
```

Listing 4.1: Semantics of all-states coverage. The listing shows a model-to-model transformation for describing the relation between the test selection criterion AllStates and the model-specific test case specification VertexTestCaseSpecification. This transformation uses metamodel elements from $MBTMM$ (Fig. 3.1) and $MBTRS_{ALL-STATES}$ (Fig. 4.13).

in several ways. For example, we modify the describes list by adding new elements to it. Since transformations can become too big to manage, Xtend allows to create user-defined libraries of functions. In the transformation for all-states, we used the two user-defined functions testModelComponents() and vertices(). The implementation of all user-defined functions is presented in Appendix B.

In the transformation in listing 4.1, we first collect all UML vertices (states and pseudo states) of all components (line 3). For each found vertex, we call the toTestCaseSpecification(Vertex) function [2]. The called function creates a new metamodel element VertexTestCaseSpecification by using the Xtend-keyword create. Inside this function we operate in the context of the newly created metamodel element so we can use the declared methods to set the referenced vertex (line 8). The final step is to add every newly created VertexTestCaseSpecification element to the describes reference (line 2).

4.5.2. All-Transitions

Definition 4.3 (all-transitions). All-Transitions coverage is achieved if every transition of the model is visited at least once.

Applying the definition for all-transitions (Def. 4.3) to our test models, full coverage is achieved when our test suite visits every transition of the UML Statechart at least once. For example, in our Online Storage Example (Fig. 4.4) the sequence of transitions t_0, t_1, t_2 gives *all-transition* coverage for the *user* component. However, depending on the Statechart,

[2]Note that Xtend allows for both *left hand* and *right hand* side notations. So the two expressions vertex.toTestCaseSpecification() and toTestCaseSpecification(vertex) mean the same.

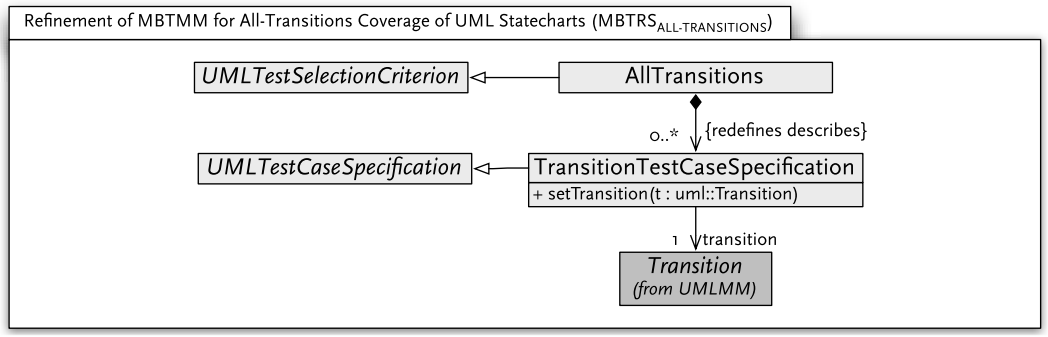


Figure 4.14.: Refinement of the common metamodel for the UML Statechart specific test selection criterion *all-transitions*. This figure refines elements from Fig. 3.1.

the sequence can become more complex. For example, *all-transition* coverage for the Client component is achieved by the following sequence:

$$t_0\ \ t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6$$
$$t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6\ \ t_1;t_3;t_5;t_6\ \ t_1;t_2$$

This corresponds to following actions in the SUT:

- client is ready (t_0) and starts with an initial *file contingent* of 10,
- the user uploads successfully 10 files ($t_1;t_3;t_5;t_6$), and each time the *file contingent* is decreased by one,
- the user tries to upload another file (t_1), but the contingent is consumed (t_2)

We model all-transitions similar to all-states as an refinement of the common metamodel (Fig. 3.1) in Fig. 4.14. The previously introduced metamodel element UMLTestSelectionCriterion is extended by the element All-Transitions. This element redefines the target of the reference describes to a new element TransitionTestCaseSpecification. It references a single, mandatory Transition element from the UML metamodel – the transition to cover.

The semantics of the all-transitions test selection criterion are described similar to all-states (listing 4.2): we first collect all UML transitions of all components (line 3). For each found transition, we call the toTestCaseSpecification(Transition) function to create a new metamodel element TransitionTestCaseSpecification. Each of the newly created elements are then added to the describes reference (line 2).

```
1 toTestCaseSpecification(mbtrs::AllTransitions allTransitions, mbtmm::TestDesign td):
2   allTransitions.describes.addAll(
3     td.testModelComponents().transitions()
4       .toTestCaseSpecification()
5   );
6
7 create TransitionTestCaseSpecification toTestCaseSpecification(uml::Transition t):
8   setTransition(t);
```

Listing 4.2: Semantics of all-transitions coverage. The listing shows a model-to-model transformation for describing the relation between the test selection criterion AllTransitions and the model-specific test case specification TransitionTestCaseSpecification. This transformation uses metamodel elements from $MBTMM$ (Fig. 3.1) and $MBTRS_{ALL-TRANSITIONS}$ (Fig. 4.14).

4.5.3. All-Configurations

UML Statecharts may have *orthogonal* states, where each region inside the state runs in parallel. To ensure that a test suite covers the possible combinations of active states arising from orthogonal states, we need special test selection criteria. Before we go into the details of the criteria, we first introduce the definition of a *configuration*:

Definition 4.4 (Configuration (Space)). A *configuration* is a snapshot of the currently active states in an UML Statechart. A configuration C_i for a Statechart is the set of states that are active at the same time. C_0 is the initial state and consecutive configurations can be found by supposing that the outgoing transitions of a configuration can be fired. The *configuration space* is the set of possible configurations of an UML Statechart.

In our Online Storage Example (Fig. 4.4), the initial configuration of the Server component is $C_0 = \{init\}$. To find the next configuration, we assume that the outgoing transition t_0 fires. This results in a new configuration $C_1 = \{main,\ main.init,\ disc,\ disc.init\}$, where $main.init$ denotes the state $init$ inside the combined state $main$. At this point, we can choose between firing the transition t_1 or t_8. While the former results in the configuration $C_2 = \{main,\ main.idle,\ disc,\ disc.init\}$, the latter gives the configuration $C_2 = \{main,\ main.init,\ disc,\ disc.spaceAvailable\}$. Following this algorithm, we get a configuration space of 16 possible configurations for the Server component.

A practical way to visualize these configurations and the transitions between them is a *reachability tree* (Masiero et al., 1994). The reachability tree is a 4-tuple $RT =< SC, T, \rightarrow, C_0 >$, with SC being the set of possible configurations, T being the set of transitions, the

transition function $\rightarrow$ being a partial function from $SC \times T$ to SC, and $C_0 \in S_C$ being the initial configuration (Masiero et al., 1994). Figure 4.15 presents the reachability tree of the Server component. It shows the 16 configurations ($C_0, C_1, \ldots, C_{15}$). In each configuration, the active state is denoted with 1 and an inactive state with 0. In the tree, a node denoted with $\uparrow C_i$ is called *history node* and represents a loop in the Statecharts configuration. The $\uparrow$ means that this node is not a terminal node and the path continues at the first occurrence of C_i. With the help of the reachability tree, we can introduce our test selection criteria for covering parallelism in UML Statecharts:

Definition 4.5 (all-configurations). All-Configuration coverage is achieved if every configuration of the model is visited at least once.

Applying the definition for all-configurations (Def. 4.5) to our test models, full coverage is achieved when our test suite visits each configuration, or in terms of the reachability tree every node of the tree, at least once. In our Online Storage Example (Fig. 4.4) the following four test cases achieve *all-configurations* coverage for the *server* component:

1. $t_0; t_1; t_2; t_6; t_7; t_8; t_9$
2. $t_0; t_1; t_2; t_6; t_8; t_9$
3. $t_0; t_1; t_2; t_8; t_9$
4. $t_0; t_1; t_8; t_9$
5. $t_0; t_8; t_9$

This corresponds to following actions in the SUT:

1.
   - server is ready and idle ($t_0; t_1$),
   - server receives a file upload request ($t_2; t_6$),
   - server authenticates the client and approves the transfer (t_7),
   - server initialized the disk space availability part and recognizes a full disc ($t_8; t_9$),
2.
   - server is ready and idle ($t_0; t_1$),
   - server receives a file upload request ($t_2; t_6$),
   - server initialized the disk space availability part and recognizes a full disc ($t_8; t_9$),
3.
   - server is ready and idle ($t_0; t_1$),
   - server receives a file upload request (t_2),
   - server initialized the disk space availability part and recognizes a full disc ($t_8; t_9$),
4.
   - server is ready and idle ($t_0; t_1$),

$$C = \{(init), (main.init, main.idle, main.receive, main.receive.init, main.receive.receivedFileReq, main.receive.receivingFile), (disc.init, disc.spaceAvailable, disc.discFull)\}$$ (states)

$$\begin{aligned} SC = {} & C_0 = \{1, (0,0,0,0,0,0), (0,0,0)\}\ C_8 = \{0, (0,0,1,1,0,0), (0,1,0)\} \\ & C_1 = \{0, (1,0,0,0,0,0), (1,0,0)\}\ C_9 = \{0, (0,1,0,0,0,0), (0,0,1)\} \\ & C_2 = \{0, (0,1,0,0,0,0), (1,0,0)\}\ C_{10} = \{0, (0,0,1,0,0,1), (1,0,0)\} \\ & C_3 = \{0, (1,0,0,0,0,0), (0,1,0)\}\ C_{11} = \{0, (0,0,1,0,1,0), (0,1,0)\} \\ & C_4 = \{0, (0,0,1,1,0,0), (1,0,0)\}\ C_{12} = \{0, (0,0,1,1,0,0), (0,0,1)\} \\ & C_5 = \{0, (0,1,0,0,0,0), (0,1,0)\}\ C_{13} = \{0, (0,0,1,0,0,1), (0,1,0)\} \\ & C_6 = \{0, (1,0,0,0,0,0), (0,0,1)\}\ C_{14} = \{0, (0,0,1,0,1,0), (0,0,1)\} \\ & C_7 = \{0, (0,0,1,0,1,0), (1,0,0)\}\ C_{15} = \{0, (0,0,1,0,0,1), (0,0,1)\} \end{aligned}$$ (configurations)

$$T = \{(t_0), (t_1, t_2, t_3, t_4, t_5, t_6, t_7), (t_8, t_9, t_{10})\}$$ (transitions)

(reachability tree)

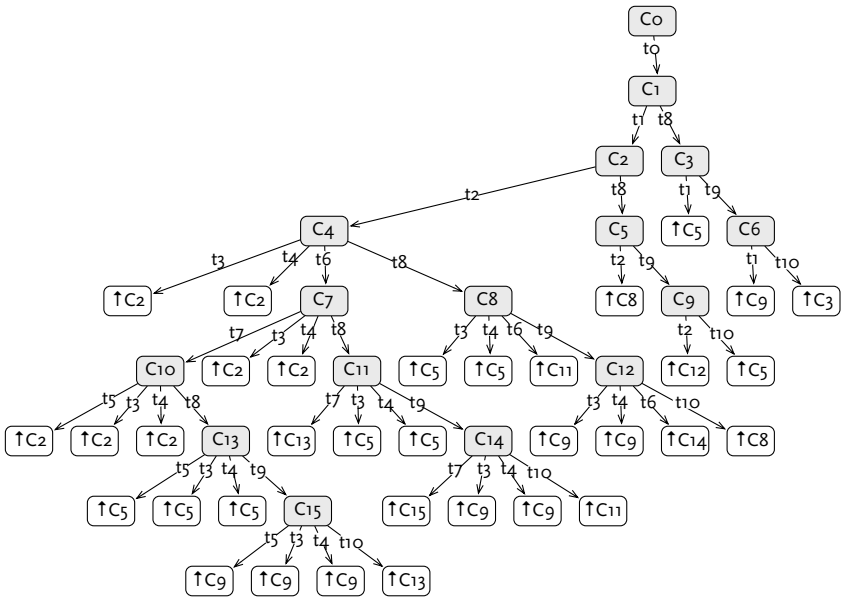


Figure 4.15.: Reachability tree for the Server component of the Online Storage Example (Fig. 4.4).

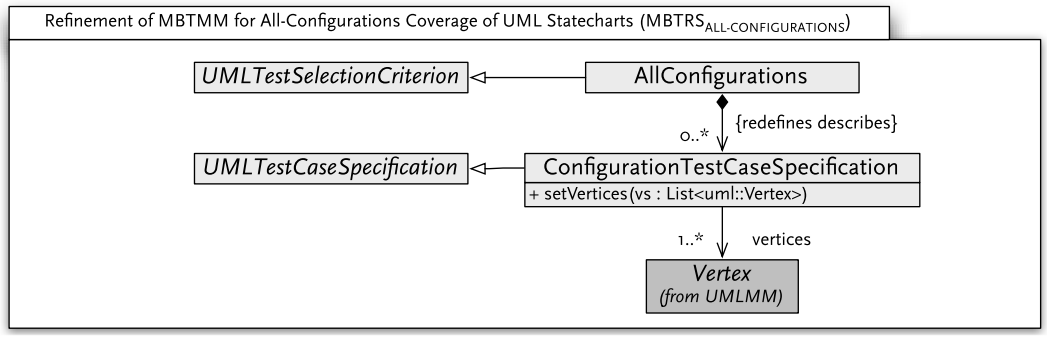


Figure 4.16.: Refinement of the common metamodel for the UML Statechart specific test selection criterion *all-configurations*. This figure refines elements from Fig. 3.1.

```
1 toTestCaseSpecification(mbtrs::AllConfigurations allConfigurations, mbtmm::TestDesign td):
2   allConfigurations.describes.addAll(
3     td.testModels().buildReachabilityTree().configurations
4       .toConfigurationTestCaseSpecification()
5   );
6
7 create ConfigurationTestCaseSpecification toConfigurationTestCaseSpecification(mbtrs::Configuration c):
8   setVertices(c.vertices);
```

Listing 4.3: Semantics of all-configurations coverage. The listing shows a model-to-model transformation for describing the relation between the test selection criterion AllConfigurations and the model-specific test case specification ConfigurationTestCaseSpecification. This transformation uses metamodel elements from $MBTMM$ (Fig. 3.1) and $MBTRS_{ALL-CONFIGURATIONS}$ (Fig. 4.16).

- server initialized the disk space availability part and recognizes a full disc ($t_8; t_9$),

5.
- server is ready (t_0),
- server initialized the disk space availability part ($t_8; t_9$),

We model all-configurations similar to all-states as an refinement of the common metamodel (Fig. 3.1) in Fig. 4.16. The metamodel element UMLTestSelectionCriterion is extended by the element All-Configurations. This element redefines the target of the reference describes to a new element ConfigurationTestCaseSpecification. It references one to many Vertex element(s) from the UML metamodel, representing the configuration of the Statechart (Def. 4.5). The semantics of the all-configuration test selection criterion are described similar to the previous test selection criteria using a model-to-model transformation. However, an intermediate step of building a reachability tree (listing 4.3) is needed: The reach-

```
1  ReachabilityTree rt = new ReachabilityTree();
2  Queue<Configuration> agenda = new Queue<Configuration>();
3
4  ReachabilityTree buildReachabilityTree(uml::Statemachine sm) {
5    Configuration initConfiguration = new Configuration(collectInitialStatesFor(sm));
6
7    enqueueNewConfiguration(initConfiguration);
8
9    while (!agenda.isEmpty()) {
10     Configuration c = agenda.dequeue();
11
12     for (Vertex vertex : c.getVertices()) {
13       for (Transition t : vertex.getOutgoings()) {
14         Configuration newConf = getNextConfiguration(configuration, vertex, t);
15         Configuration historyConf = rt.getConfiguration(newConf);
16
17         if (historyConf != null) {
18           newConf = historyConf;
19         } else {
20           enqueueNewConfiguration(newConf);
21         }
22
23         ConfigurationTransition ct = createConfigurationTransition(rt, c, t, newConf);
24         rt.getConfigurationTransitions().add(ct);
25         c.getOutgoings().add(ct);
26       }
27     }
28   }
29   return rt;
30 }
31
32 void enqueueNewConfiguration(mbtrs::Configuration c) {
33   agenda.add(c);
34   rt.getConfigurations().add(c);
35 }
```

Listing 4.4: Pseudo code of the algorithm to build a reachability tree using a classical breadth-first search.

ability tree is built using a classical breadth-first algorithm. Listing 4.4 shows the pseudo code of our algorithm (the missing functions of the algorithm can be found in the Appendix C). The shown function buildReachabilityTree takes an UML Statechart and builds the reachability tree for it. It uses a queue data structure (variable agenda) to store intermediate configurations and transitions between configurations as it traverses the configuration space, as follows:

1. collect the first configuration using the initial states of the Statechart, and enqueue this configuration (line 7)
2. dequeue a configuration (line 10) and examine it:
   - determine the set of successors (the direct child configurations) by traversing every outgoing transition of every state in the dequeued configuration (line 14)
   - If the child configuration is a new configuration (not a already known history node), then enqueue it
   - If the child configuration is a new configuration (not a already known history node), then enqueue it
3. If the queue is empty, quit and return the resulting reachability tree (line 29)
4. If the queue is not empty, repeat from step 2

The model-to-model transformation for *all-configurations* is completed by converting the configurations of the resulting reachability tree into ConfigurationTestCaseSpecification elements (listing 4.3, line 7 to 9).

4.5.4. All-Configuration-Transitions

Definition 4.6 (all-configuration-transitions). All-Configuration-Transitions coverage is achieved if every outgoing transition of a configuration, which results in a new configuration, is visited at least once.

Applying the definition for all-configurations (Def. 4.6) to our test models, full coverage is achieved when our test suite visits every transition in the reachability tree at least once. Due to space considerations, we skip showing the full test suite of the Online Storage Example. However, such a test suite can easily be created for the server component using the reachability tree in Fig. 4.15 by constructing test cases which cover all edges of the tree.
We model all-configuration-transitions similar to all-transitions as an refinement of the common metamodel (Fig. 3.1) in Fig. 4.17. The metamodel element UMLTestSelectionCri-

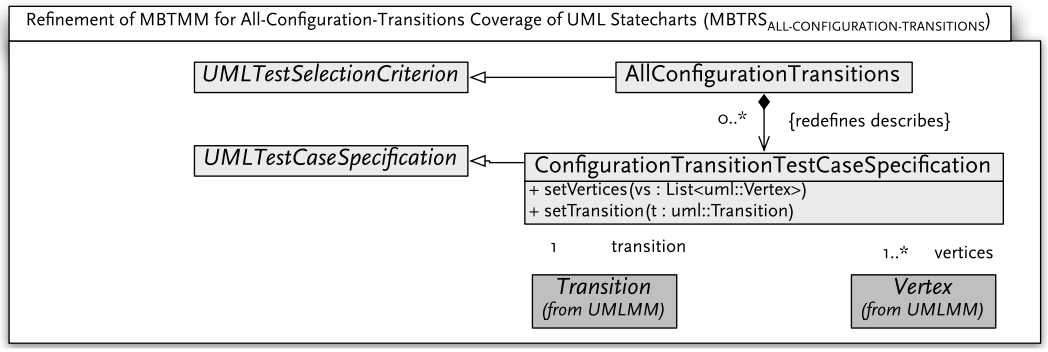


Figure 4.17.: Refinement of the common metamodel for the UML Statechart specific test selection criterion *all-configuration-transitions*. This figure refines elements from Fig. 3.1.

terion is extended by the element All-Configuration-Transitions. This element redefines the target of the reference describes to a new element ConfigurationTransitionTestCaseSpecification. It references one to many Vertex element(s) from the UML metamodel, representing the configuration of the Statechart (Def. 4.5), and a mandatory Transition – the transition between two configurations to cover.

The semantics of the all-configuration-transitions test selection criterion are described similar to all-configurations (listing 4.5): The previously built reachability tree using our breadth-first algorithm is reused. For all resulting configurations (line 3), the outgoing transitions are collected (line 8). For every of these transitions, a ConfigurationTransitionTestCaseSpecification, referencing the set of UML vertices and the UML transition, is created (line 10 to 13).

4.5.5. Explicit Test Case Specifications

In addition to derive test case specifications using test selection criteria (like *all-states*, *all-transitions*, ...), it may be necessary to manually formulate additional test case specifications. These so called *explicit test case specifications* may be used to restrict the test case generator to use specific paths through the test model, focus the testing on common use-cases, or transform high-level requirements into test case specifications (Utting and Legeard, 2006). To give an example, let's recapitulate the presented test suite covering *all-configuration* of the server component of the Online Storage Example (Sec. 4.5.3). Although the resulting tests cover the criterion fine, the tests always end with the sequence $t_8; t_9$, meaning the

```
toTestCaseSpecification(mbtrs::AllConfigurationTransitions allConfigurationTransitions, mbtmm::TestDesign td):
  allConfigurationTransitions.describes.addAll(
    td.testModels().buildReachabilityTree().configurations.
      toConfigurationTransitionTestCaseSpecification()
  );

toConfigurationTransitionTestCaseSpecification(mbtrs::Configuration c):
  c.outgoings.toConfigurationTransitionTestCaseSpecification(c);

ConfigurationTransitionTestCaseSpecification toConfigurationTransitionTestCaseSpecification
  (mbtrs::ConfigurationTransition ct, mbtrs::Configuration c):
    setVertices(c.vertices) ->
    setTransition(ct.referredTransition);
```

Listing 4.5: Semantics of all-configuration-transitions coverage. The listing shows a model-to-model transformation for describing the relation between the test selection criterion AllConfigurations and the model-specific test case specification ConfigurationTransitionTestCaseSpecification. This transformation uses metamodel elements from $MBTMM$ (Fig. 3.1) and $MBTRS_{ALL-CONFIGURATION-TRANSITIONS}$ (Fig. 4.17).

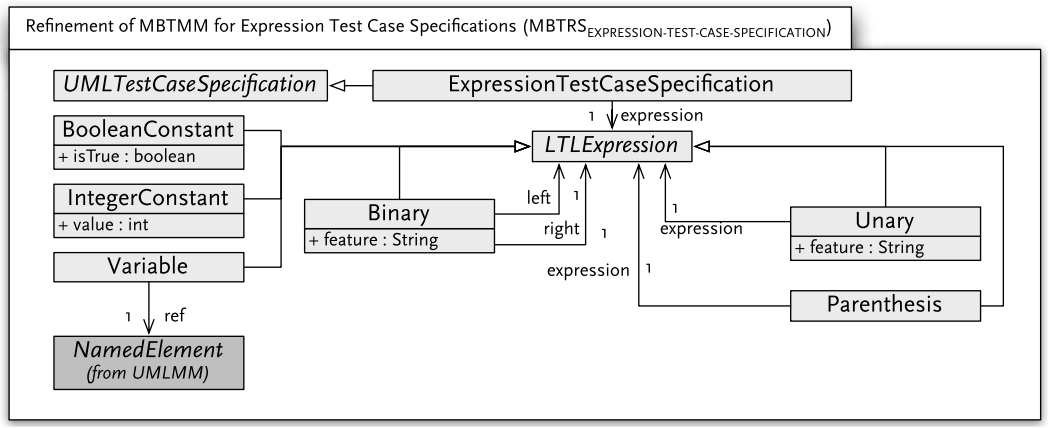


Figure 4.18.: Refinement of the common metamodel for LTL expression based explicit test case specifications. This figure refines elements from Fig. 3.1.

initialization of the disc space and recognition of the full disc. Unfortunately, this test suite does not cover many of the interesting paths of the SUT, like the successful upload of a file. To enhance the resulting test suite with these paths, we could combine the *all-configuration* criterion with another criterion, like *all-transitions*. However, this might result in a unnecessary big test suite. As an alternative, we can combine the *all-configuration* criterion with one of the presented *test case specifications*, like TransitionTestCaseSpecification, to explicitly cover transition t_5 (completion of the upload transfer). This combination of generated and manual test case specifications is a powerful tool to reduce the size of the resulting test suite to the more interesting behaviors of the SUT. Going further, even the manual addition of test case specifications might not be sufficient to force the test generator to include the desired behavior. In the Online Storage Example, we might be interested in a scenario where the user uploads a file (state $receivingFile$), but immediately cancels the upload (transition t_3). This scenario cannot be easily described with the presented test case specifications. Instead, we need a more expressive way to describe the scenario. For this situation, we rely on existing work (Ammann et al., 2001; Tan et al., 2004; Fraser and Wotawa, 2008b) where temporal logics are used as explicit test case specification to drive the test generator. Using LTL formulae, for example, we can express *safety*, *liveness*, or a combination of safety and liveness properties of our test model for the purpose of generating tests for the SUT. When using LTL for verification of a system using modern model checkers, the model is typically described using a set of variables (Fraser, 2007). Consequently, atomic propositions of the LTL formula are defined over these variables. However, in our approach, we use LTL for guiding the test generation process using UML Statecharts, so atomic propositions have to be defined over UML elements. To illustrate this approach, we show an explicit test case specification for the mentioned scenario, where the user uploads file, but cancels the upload. For this scenario, the corresponding LTL formula is $G(receivingFile)$ & $F(t_3)$, which states that whenever the state $receivingFile$ is active, some time later the transition t_3 has to be fired.

We model explicit test case specifications as an refinement of the common metamodel (Fig. 3.1) in Fig. 4.18. The metamodel element UMLTestCaseSpecification is extended by the element ExpressionTestCaseSpecification. This element references one LTLExpression element, which serves as an abstract super-type of all other expressions. Together with all sub-types of LTLExpression, the abstract syntax tree of an LTL expression can be built. The most interesting sub-type is Variable, which implements our requirement to reference

UML elements from our formula. In the UML metamodel, the referenced NamedElement represents elements that may have a name. It is the super-type of all structural elements of a UML Statechart (like (pseudo an combined) states, transitions, events etc.). The concrete syntax for our LTL expressions (see Appendix D) also allows mathematical expressions over variables. The semantics of these expressions are defined by means of model-to-model transformations in the next chapter, where we present our approach for the test generation using model checkers (Chap. 4.6).

4.6. Test Case Generation

In the last sections, we presented the refinement of the structural parts of our formalization, naming the test models, test suites, and test selection. To reach our goal to automate the test generation, we present the formalization of the test generation process of our refinement. We use following steps to show our approach:

- We propose a common workflow for MBT using model checkers with tasks inside.
- We discuss each task of this workflow and show how these tasks contribute to the final goal.
- Although the common workflow contains mostly abstract tasks with various possible implementations, we show for every task a default implementation as a proof of concept.

Testing with Model Checkers

Automatic test generation using model checkers was initially proposed by (Callahan et al., 1996) and (Engels et al., 1997). The idea is to create an abstracted model of a SUT with the aim of generating test cases. This model will contain the expected behavior of the SUT, and thus solves the *oracle* problem. Model checking has been invented as a tool for formal verification (Queille and Sifakis, 1982; Clarke et al., 1986). A model checker accepts a model in an automaton-like specification, and a property of the system described, in general, with a temporal logics language. The model checker then explores the state space and tries to show that the system has the described property. If it fails to show the property, it creates a *counterexample* which typically shows a path in the state space which leads to a state where the property is violated. An analyst can then take this counterexample to find the problem in the original specification of the system, or the model itself.

The main idea for testing with model checkers is to automatically interpret counterexamples as test cases (Fraser et al., 2009). The challenge is to force the model checker to systematically generate these counterexamples according to (a set of) test requirements. A common approach for this is to use *trap properties* (Dias Neto et al., 2007). Trap properties are the *negated* version of a test requirement, or test selection criterion in our case. The model checker is asked to verify the trap property, for example that a certain state cannot be reached. If the model checker finds a counterexample for the trap property, the counterexample is interpreted as a test case that satisfies the test requirement.

Some of the often used model checkers for test case generation are *simple promela interpreter* (SPIN) (Holzmann, 1997), *symbolic analysis laboratory* (SAL) (Moura et al., 2004), and the symbolic model checker NuSMV (Cimatti et al., 1999). Most of the model checkers support CTL and LTL for model checking. Although many problems of automatic test generation using model checker have been solved, some of the problems remain and are subject to active research (Fraser et al., 2009). In addition, some optimization approaches have been proposed:

- **test suite minimization**: test suite minimization, an approach not specific to testing with model checkers, tries to reduce the redundancy resulting from generating test cases – for example identical test cases. This can help when resources for testing are limited, e.g. when it is not possible to execute all of the test cases when there are too many. Minimization deals with removing or combining test cases so that the size of the test suite is reduced, but not the overall coverage. In the context of testing with model checkers, several approaches to this idea have been proposed (Heimdahl and George, 2004; Zeng et al., 2007).
- **monitoring**: Another optimization approach, called monitoring, tries to avoid the generation of test cases at all. While test suite minimization deal with a post-processing of already generated test cases, the monitoring approach tries to detect, during the process of test generation, which temporal formulas are already satisfied by the current set of test cases. For any already satisfied formula, the generation is skipped, which avoids the (potentially computation expensive) generation of test cases. For testing with model checkers, a given test case has to be checked (monitored) against all the remaining temporal formulas to see which formula the test satisfies or violates. Several approaches to this idea have been proposed (Fraser and Wotawa, 2007b; Artho et al.; Arcaini et al., 2013a).

- **ordering of test case specifications**: When monitoring a test case generation, the order of the test case specifications might have an influence on the size of the resulting test suite (Fraser et al., 2009). In the worst case ordering, each test case covers only one test case specification, so monitoring has no effect. Trying to optimize the order of the test case specifications might therefore help to further reduce the redundancy of test suites. In the context of testing with model checkers, several approaches for ordering test case specification have been proposed (Fraser and Wotawa, 2008a; Hong and Ural, 2005).

Common Workflow For Automatic Test Generation With Model Checkers

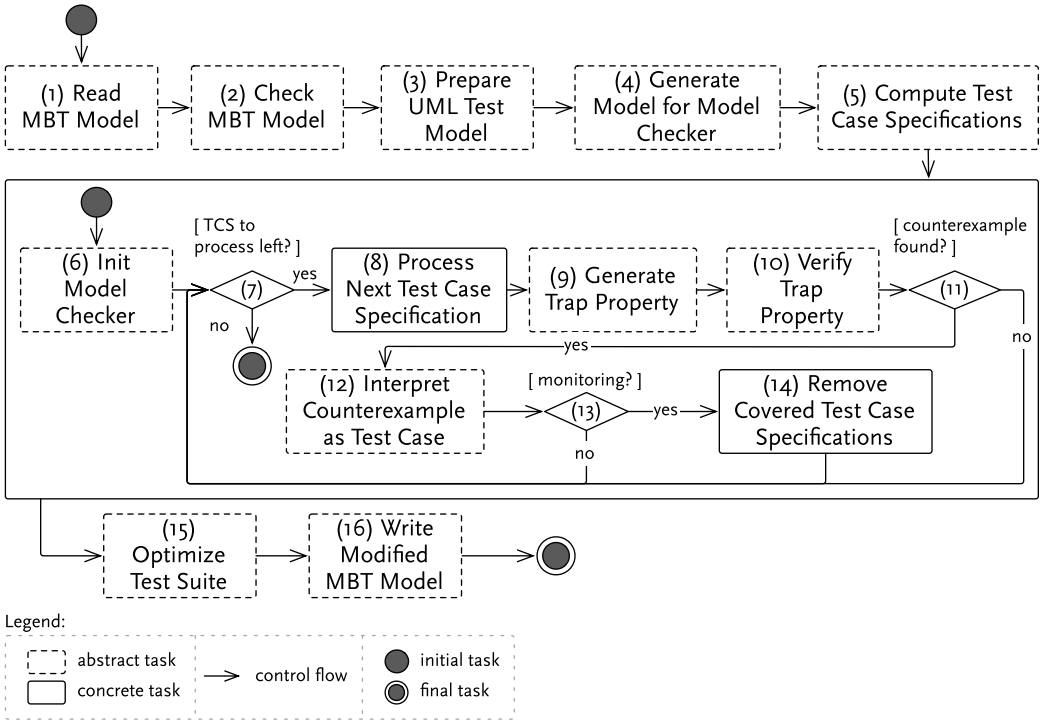


Figure 4.19.: Abstract workflow for automatic test generation with model checkers. The workflow consists of abstract and concrete tasks. It is divided into three main areas: steps 1 to 5, 6 to 14, and 15 to 16.

Based on the above observations about the typical steps for testing with model checkers, together with the presented optimization steps, we propose a common abstract workflow (Def. 3.2) for automatic test generation with model checkers in figure 4.19. This workflow consists

of both abstract and concrete tasks. It contains three main parts: the first part (steps 1 to 5) deals with reading and preparing a model of the *MBTRS* metamodel (Fig. 4.12), conversion of the UML Statechart to a input model for the model checker, and the computation of the test case specifications from test selection criteria. The second part (steps 6 to 14) represents the heart of the whole process – the automatic test generation – using a loop to convert test case specifications to test cases using the model checker. The third part (steps 15 to 16) deals with optimization and persistence.
In the following, we go into the details of this workflow by discussing the intent of each step. To show the feasibility of the abstraction, for each abstract task, we present a default concrete implementation based on existing approaches of the MBT community.

Step 1 to 3: Read, Check And Prepare UML Test Model

The first step of the workflow is to read a model of the *MBTRS* metamodel (Fig. 4.12). This model contains four main parts:

- A reference to the workflow and its tasks which have to be executed as described in chapter 3.
- A reference to an UML based test model as described in chapter 4.3.
- The definition of the test selection criteria for the test generation.
- The definition of any additional explicit test case specifications for the test generation.

A typical implementation of this step is to read the model from a storage, i.e. file or database storage, and to keep this model in the computer's memory for further processing. In our prototypical implementation (Sec. 4.7) we use a textual concrete notation for the *MBTRS* and use existing EMF based tools for reading this model.

In the second step, we propose a consistency check of the read model. Although the syntactic correctness of models is typically assured by tools, additional semantic constraints have to be checked. In case of our UML based approach, we let the test designer create the UML based test model with existing UML tools for graphical editing. While this improves the usability for the user, we have to cope with the fact that UML tools are meant for general purpose modeling, and allow the user to create arbitrary test models which are not based on our formalization. In our implementation of this task, we integrated the following checks:

- Is any test selection criterion or explicit test case specification given?
- Can the referenced UML model be found?

- Does the UML model use our notation for classes, state machines, variables, guards and actions?

After reading the model and checking it for consistency, we propose an additional preprocessing step (step 3) to improve the quality of the UML test model and bring it to a normalized form. In this step, some basic transformations are typically undertaken. As an example, in our formalization, we allow the test designer to use the *else* notation in decision states to express a choice where none of the other choices match. However, although the *else* notation is a user friendly shorthand, we have to replace any *else* expression with the corresponding concrete expression for further processing. For example, we used the *else* notation in the client component of our Online Storage Example (Fig. 4.4, see transitions t_2 and t_3) to express a different expected behavior when the user has a sufficient file contingent (variable fileContingent), or not. The UML choice has two transitions, with one of the having the guard $fileContingent > 0$ and the other using the $else$ expression. In this simple example, the $else$ expression has to be transformed to the explicit form $!(fileContingent > 0)$. Another example for necessary transformations are problems with naming. In our approach we always refer back into UML elements from our test case specifications and test cases. Since all used UML elements inherit from the UMLNamedElement, we rely on these names for referencing. However, as we allow the test engineer to create the test model using general purpose UML tools, she might simply forget to assign names to every element, which makes it impossible to reference these elements. To overcome this problem, we decided to simply generate elements names, where they are missing. For both the *else* expression and naming generation, we applies a model-to-model transformation of the UML test model.

Step 4: Generate Model for Model Checker

The goal of the fourth step of the workflow is to transform the UML based test model to an input model of the model checker. This transformation will in general be a model-to-text transformation, since most of the model checkers support a textual notation. The most challenging problem of this transformation is to correctly map the UML concepts and semantics to a model checking problem. In any implementation of this transformation, all of the presented concepts of our test model (Sec. 3.2, Sec. 4.3) have to be supported and mapped. As this transformation highly depends on the concepts of the used model checker, we continue this section using a specific model checker.

In our implementation of this step, and further this dissertation, we use NuSMV (Cimatti et al., 1999) as the model checker to automatically generate test cases. The decision on NuSMV was mainly motivated by the good possibilities of technical integration with our technologies. For example, NuSMV can generate counterexamples in a XML format, which we easily can interpret as a model of an ecore metamodel. However, our approach is not tied to NuSMV [3], since we see a model checker as a *black-box tool* to actually convert our test case specifications to test cases. NuSMV supports a module concept, allows defining variables, and implements concurrent execution of the declared modules. Details on the NuSMV syntax and the supported language concepts can be found in (Cimatti et al., 1999). The structural overview to the generated NuSMV model is shown in figure 4.20.

Our test model is described using an UML component for defining the structural part of our SUT, and an UML Statechart for the behavioral part. In addition, communication between components is modeled by declaring an UML Interface with signals and method, and referencing the other component (Sec. 4.3). In our transformation to NuSMV, we create a corresponding *MODULE* definition for each UML Component (line 1 to 78). If the UML Component communicates with other components, the NuSMV module gets a reference to the other NuSMV module (line 1). The modules are structured the following way:

- Declaration of variables and macros (line 1 to 19)
- Initialization variables (line 21 to 35)
- Consumption and production of events (line 37 to 57)
- Calculation of the next step of the execution (line 59 to 78)

In the declaration part (line 1 to 19), we define the variables for the NuSMV module. Since our test models can run independently and communicate through signals, we added a signal (first in first out) queue into the model (line 4), because NuSMV does not include a queue implementation natively. The queue is modeled as an array. Its size is a configurable parameter (BUFFER-SIZE) and defaults to zero, which in fact allows *synchronous communication*. The entries of the array are an enumeration of the possible input signals of the UML component, where nil indicates an empty slot of the queue. Next, we transform each variable of the UML component to a corresponding NuSMV variable (line 6 to 9). Value ranges of integer variables (4.3) are transformed into the NuSMV format (line 6). To transform the behavioral part of our test model, namely the UML Statechart, we map the region, states, and transitions of the Statechart into NuSMV variables (line 10 to 14). For each region of

[3] As we later show, our framework has been successfully used by third parties to use the SPIN model checker.

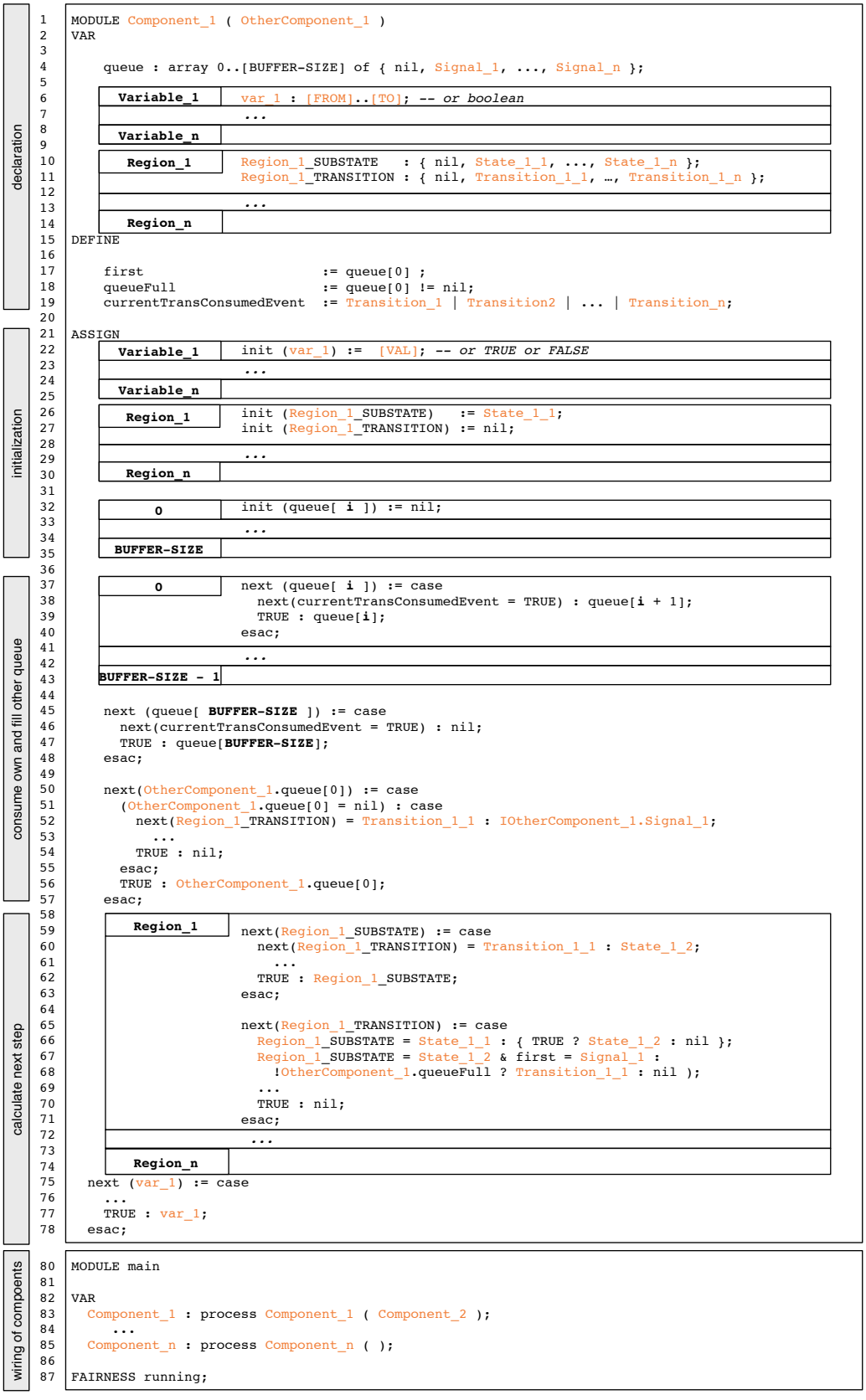


Figure 4.20.: Structure of the generated NuSMV model. Figure arrangement is inspired by (Cichos, 2013).

the Statechart, also nested regions inside complex states, we enumerate the states and transitions inside that region. The nil value indicates that no state or transition is active in the region. The rest of the declaration part contains definitions of some shorthand macros (line 15 to 19).

In the initialization part (line 21 to 35), we first initialize the variables based on the initial values of the UML class variables (line 22 to 24). Then, we set initial values for the state and transition of every region (line 26 to 30). The initial state is set based on the initial state of the UML Statechart. If a region is not active in the first step, the regions state is set to nil, denoting that no state is active in this region. Analogous, the initial transition, if any, is set to the regions transition variable. Finally, we initialize the queue with nil values in every slot (line 32 to 35).

The next part of the NuSMV model deals with consumption of the queue, and placement of signals to queues of other connected components (line 37 to 57). Here we first define the semantics of the event consumption. When the next active transition consumes an event, we shift all the events by one slot (line 37 to 43). The last slot needs special attention in this case since it becomes empty (line 45 to 48). After the consumption part, we generate code for sending signals to other connected components. When the input queue of the connected component is empty and the next active transition is a transition with outgoing signals, we set the first slot of the incoming queue of the connected component (line 50 to 57).

The next part implements the UML step semantics for changing active states and transitions, and the changing of variables (line 59 to 74). For each region in the Statechart, the next active transition, if any, is determined. Based on that transition, the next active state is calculated (line 59 to 62). If no transition is active, the state remains at the current value (line 62). If the region is left by the next active transition, the region variable becomes nil. After calculating the next state, we calculate the next active transition for each region (line 65 to 71). If the transition can be chosen non-deterministically (Cap. 4.3), we use a special NuSMV syntax construct (using curly braces) to allow the non-deterministic choice whether to activate the transition, or not (line 66). If the next transition fires as a result of an incoming signal, the calculation is guarded with that signal (line 67). Finally, if the next active transition would send a signal to another connected component, an additional guard is added that the input queue of the other component is not empty, so the next transition can send its signal (line 68). If no transition becomes active, the transition vari-

able becomes nil. The last calculation deals with changes to variables in the next step (line 75 to 78). Variables can be modified in various ways, depending on their type (boolean or integer). Transitions can change variables based on the output action, or *free* variables can be assigned by the model checker non-deterministically.

During the model-to-text transformation, the above structure is repeated for all components of the test model. To give the NuSMV model checker a starting point, a final module named main is created (line 80 to 85). This module instantiates all the other modules, and injects the module references, if necessary (line 83). To allow a non-deterministic execution of all modules, we used the NuSMV *process* feature, which treats each module as an independently executed process. Finally, we tell the NuSMV model checker to schedule all modules *fair*, which means that the model checker restricts the attention only to *fair execution paths* (Cimatti et al., 1999).

Step 5: Compute Test Case Specifications

In the fifth step of the workflow the test selection criteria of the test model have to be converted to test case specifications. Implementations of this step, in general, will use model-to-model transformations, as we presented in section 4.5. In our implementation, we support all of the presented test selection criteria.

Step 6 to 14: Initialization And The Main Test Generation Loop

After generating the input model for the model checker and computing test case specification from the test selection criteria, we come to the test generation part of the workflow. First, we deal with initializing the model checker (step 6), so we can use it for test generation. The test generation itself is modeled using a loop (steps 7 to 14) over all test case specifications (outcome of step 5). In this loop, we first check if any test case specification is left to process (step 7). If not, we finish the test generation and proceed with the post-processing steps (15 to 16). If any test case specification is left, we generate a trap property for it (step 9), and let the model checker find a counterexample (step 10). In case that no counterexample could be found, this test case specification cannot be satisfied and thus no test could be generated for it. We proceed with the next test case specification. However, if the model checker was able to find a counterexample, we interpret the counterexample as a test case (step 12). If *monitoring* is enabled, we remove all covered test case specification for the input list (step 14). If monitoring is not enabled, we process with the next test case

specification.

Step 6: Initialize Model Checker

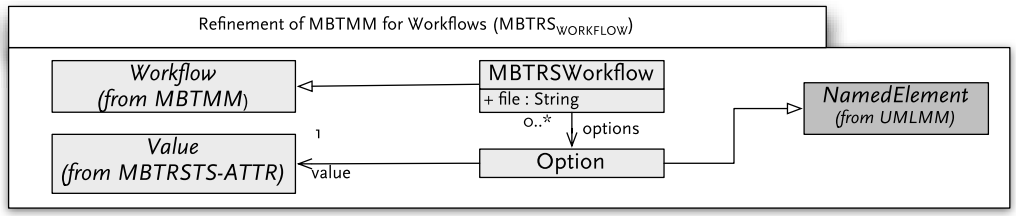


Figure 4.21.: Refinement of the Workflow concept. The new element MBTRSWorkflow references a list of options which are used to set parameters of a model checker. This figure refines elements from Fig. 3.1.

In this step, the model checker needs to be initialized with the generated model (step 4). For this, the implementations typically have to use an external interface of the model checker. Any additional model checker specific actions (like settings additional parameters) have to be done in this step. In order to give the test engineer control over the initialization of the model checker, any initialization parameter of the used model checker should be settable in the test model. We followed up this idea with an extension of the basic formalization of the Workflow concept in Fig. 4.21. In this extension, we allow to attach a list of typed options to a workflow. These options can then be used by an implementation of this sixth step to initialize the model checker. Another possible application for this extension is setting parameters for the code generation (see step 4), where we exemplary can set the buffer size of the NuSMV modules.

In our implementation of this step, we had to use or build an interface to the NuSMV model checker. As we will show in the next section (Sec. 4.7), we wrapped the model checker with an JNI Interface, so that every command of the model checker becomes usable to our implementation and initialization parameters could be set.

Step 9: Generate Trap Property

The goal of this step is to convert the test case specifications to trap properties. Depending on the used model checker, this means a model-to-text conversion to LTL, CTL, or any other language supported by the model checker to express properties. In our implementation of

this step, we implemented model-to-text transformations for all of the presented test case specifications in Sec. 4.5 in figure 4.22. The transformations take an instance of a test case specification model element and convert them to a negated CTL property. For example, to force the model checker to find a counterexample for a VertexTestCaseSpecification, we state that there exists no path where the referenced variable becomes true (Fig. 4.22(a)). This desire is expressed using the property $AG!(v1)$, where $v1$ references a UML Vertex. The other test case specifications are transformed in an analogous manner. However, some trap properties are more complicated to express than others. The ConfigurationTransitionTestCaseSpecification for example references a set of vertices (the configuration) and a transition which is active. Transforming this specification to a trap property needs to operate both on the current state, and on the next state (Fig. 4.22(d)).

Step 10: Verify Trap Property

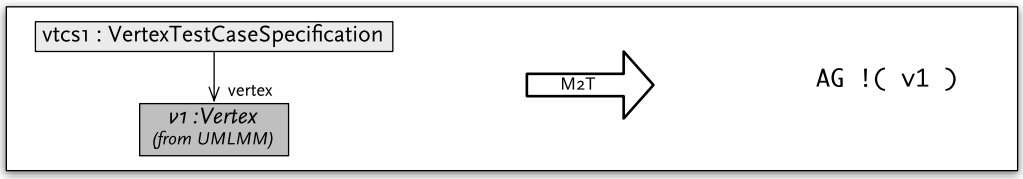
In this step the, the model checker is asked to verify our previously generated trap properties (step 9). The external interface to the model checker has to be extended to support automatic verification, and also needs to process the result of the model checker. The result of the verification may be a counterexample, representing a test case for the original test case specification.

In our implementation of this step, we extended our JNI interface to NuSMV to be able to verify our generated trap properties. NuSMV is able to produce counterexamples in various formats, ranging for a human-readable text format to machine processable XML documents. For our implementation, we chose the XML format to be able to automatically process the counterexample (see step 12).

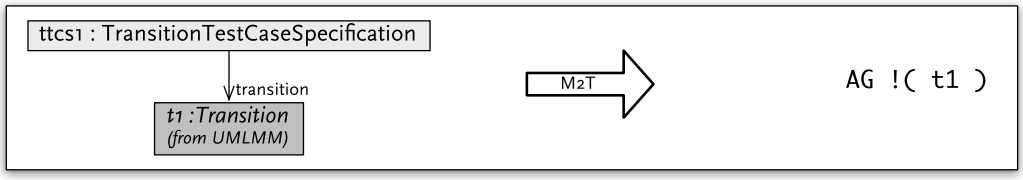
Step 12: Interpret Counterexample As TestCase

If a counterexample was generated due to the verification of a trap property (step 10), we have to parse the counterexample and interpret its content as a test case. Our trap properties are negations of our root requirement (our test case specification). The generated counterexample therefore shows one possible path in the state space where this requirement can be shown in the SUT. Any implementation of this step has to create instances of our test metamodel defined by $MBTRS_{TS}$ (Fig. 4.6), $MBTRSTS_{ATTR}$ (Fig. 4.8), and $MBTRSTS_{STRUCT}$ (Fig. 4.10).

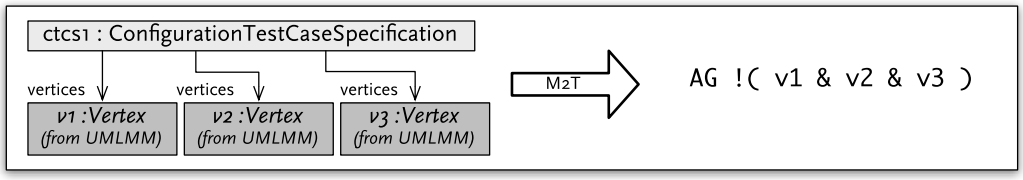
In our implementation of this step, we leverage on XML based counterexamples of



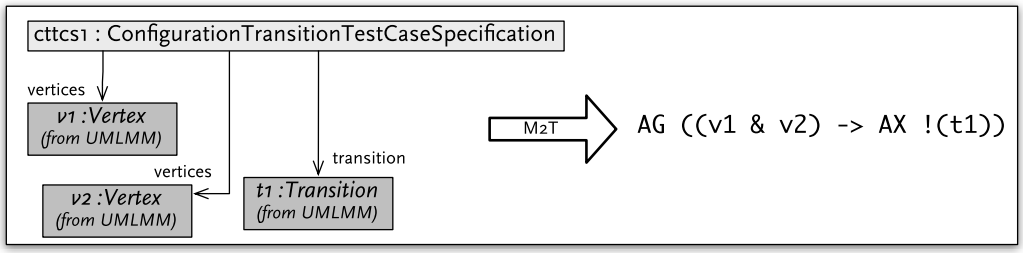
(a) Transformation of an exemplary VertexTestCaseSpecification.



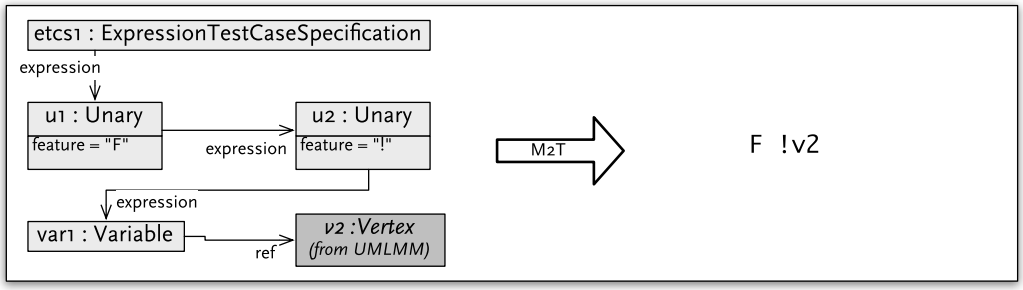
(b) Transformation of an exemplary TransitionTestCaseSpecification.



(c) Transformation of an exemplary ConfigurationTestCaseSpecification.



(d) Transformation of an exemplary ConfigurationTransitionTestCaseSpecification.



(e) Transformation of an exemplary ExpressionTestCaseSpecification (written in LTL).

Figure 4.22.: Overview of the implemented model-to-text transformations for the presented test case specifications of Sec. 4.5. The models are converted into CTL trap properties to be used with NuSMV.

```
<counter-example>
    <node>
      <state id="1">
        <value variable="{NAME}">{VALUE}<value>
      </state>
      <input id="2">
        <value variable="{NAME}">{VALUE}<value>
      </input>
    </node>
    <node>
      <state id="2">
        <value variable="{NAME}">{VALUE}<value>
      </state>
      <input id="3">
        <value variable="{NAME}">{VALUE}<value>
      </input>
    </node>
    ...
</counter-example>
```

Listing 4.6: Structure of the XML based counterexample of NuSMV.

NuSMV. Listing 4.6 shows the structure of the NuSMV counterexample XML file. It contains a list of node elements, with each of them representing a state in the state space. In each node element, NuSMV outputs the assignments of all variables defined by the source model. It distinguishes between input elements, which contain information about the scheduling of components, and state elements, which contain the variable assignments of all components.

To interpret this XML file we have to recap how we generated the initial source model for NuSMV (see step 4). We encoded all components of the UML based test model, together with their variables. In addition, we encoded the Statechart's regions, states and transitions. The counterexample generated by NuSMV will contain all these information. Using the visualization in figure 4.23, we interpret the contents of the XML file the following way:

- For each node element we create an instance of the UMLTestStep metamodel element (see $MBTRS_{TS}$, Fig. 4.6)
- In the input element, NuSMV introduces a special variable running which marks the active (scheduled) component in this step. The active component has the value TRUE, and we interpret this by setting the reference scheduledComponent of the previously created UMLTestStep metamodel element.

- In the state element, we interpret the variable assignments of the currently scheduled component and create a AttributeValueOutput metamodel element for each of them (see $MBTRS_{ATTR}$, Fig. 4.8)
- The state element also contains the information about which UML states and transitions are active in this step. We interpret that by creating ConfigurationOutput and TransitionOutput metamodel elements respectively. (see $MBTRS_{STRUCT}$, Fig. 4.10)
- For any of the currently active UML transitions, we create ReceiveSignalEventInput metamodel elements if the transition is triggered by a signal. Analogously, we create SendSignalActionOutput metamodel elements if the transition emits a signal. (see $MBTRS_{TS}$, Fig. 4.6)

Figure 4.23.: Interpretation of the XML based counterexamples of NuSMV as test cases. Figure arrangement is inspired by (Cichos, 2013).

Step 14: Remove Covered Test Case Specifications

To allow the *monitoring* optimization approach (Sec. 4.6), we introduce a new workflow configuration parameter monitoring (boolean). If this parameter is set to true, we remove all test case specifications that are covered by the current test case. This coverage information has to be built up for every test case. Our approach is to check which test case specifications are covered by the generated test case. Listing 4.7 shows the pseudo code of a naïve implementation for this task. For every test case specification, we check in every step of the generated test case if the test case specification is covered. If it is covered, we add this test case specification to the *satisfies* relation of the TestCase metamodel element. The actual coverage checks for the presented test case specification are also shown in this listing. The checks basically follow an similar approach in that they look for the existence of a particular *output* type in the test step, i.e., TransitionOutput and ConfigurationOutput and compare this output to the current test case specification, i.e., TransitionTestCaseSpecification or ConfigurationTestCaseSpecification.

With this approach we enhance the model with coverage information by maintaining references between TestCases and TestCaseSpecifications. Having these information, the implementation of the actual *monitoring* is trivial, since we now can iterate over all covered test case specifications of a test case, and remove each of them from the original list of test case specifications. Listing 4.8 shows the pseudo code of our implementation.

Step 15: Optimize Test Suite

After all test cases have been generated we might want to apply a *test suite minimization* approach to remove redundancies in the test suite. Any implementation of this step may be implemented as a model-to-model transformation which reduces the number of test cases, or even test steps. As noted before, test suite minimization is subject to active research and several approaches have been proposed (Heimdahl and George, 2004; Zeng et al., 2007).

We choose a very basic optimization in our implementation. The basic idea is to remove all test cases which do not contribute to the overall coverage. A test case contributes to the coverage if it has a *unique* coverage of a test case specification. A coverage is *unique* if the test case specification is only covered by one test case. Removing this test case would therefore reduce the overall coverage. On the other hand, if a test case does not have such a *unique* coverage, we safely can reduce this test case without affecting the overall coverage[4].

[4]Note that removing any test case from a test suite has been shown to reduce the overall fault detection ability,

```
addCoverageLinks (testCase ∈ MBTMM, testDesign ∈ MBTMM) {
  foreach testCaseSpecification in testDesign.testCaseSpecifications {
    foreach testStep in testCase.testSteps {
      covered = checkCoverage(testStep, testCaseSpecification)
      if(covered) {
        testCase.satisfies.add(testCaseSpecification)
      }
    }
  }
}

boolean checkCoverage(testStep ∈ MBTMM, vertexTestCaseSpecification ∈ MBTRS_TS) {
  foreach activeVertex in testStep.outputs.filterBy(ConfigurationOutput).activeVertices {
    if(activeVertex == vertexTestCaseSpecification.vertex) {
      return true
    }
  }
  return false
}

boolean checkCoverage(testStep ∈ MBTMM, transitionTestCaseSpecification ∈ MBTRS_TS) {
  foreach activeTransition in testStep.outputs.filterBy(TransitionOutput).activeTransitions {
    if(activeTransition == transitionTestCaseSpecification.transition) {
      return true
    }
  }
  return false
}

boolean checkCoverage(testStep ∈ MBTMM, configurationTestCaseSpecification ∈ MBTRS_TS) {
  return configurationMatches(testStep, configurationTestCaseSpecification.vertices)
}

boolean checkCoverage(testStep ∈ MBTMM, configurationTransitionTestCaseSpecification ∈ MBTRS_TS) {
  if(!configurationMatches(testStep, configurationTransitionTestCaseSpecification.vertices)) {
    return false
  }
  return testStep.next.filterBy(transitionOutput ∈ MBTRS_TS)
    .activeTransitions.contains(configurationTransitionTestCaseSpecification.transition)
}

boolean configurationMatches(testStep ∈ MBTMM, vertices ∈ UMLMM) {
  return testStep.outputs.filterBy(ConfigurationOutput).activeVertices.containsAll(vertices)
}
```

Listing 4.7: Pseudo code for a naïve implementation to find covered test case specifications.

```
removeAlreadyCoveredTestCaseSepcifications (testCaseSpecification ∈ MBTMM,
  testCase ∈ MBTMM, testDesign ∈ MBTMM) {

  foreach testCaseSpecification in testCase.satisfies {
    testDesign.testCaseSpecifications.remove(testCaseSpecification)
  }
}
```

Listing 4.8: Pseudo code for removing already covered test case specifications from the list of the test generator (called monitoring).

```
optimizeTestSuite(testSuite ∈ MBTMM) {
  foreach testCase in testSuite.testCases {
    unique = false;
    foreach testCaseSpecification in testCase.satisfies {
      if(testCaseSpecification.satisfiedBy.size == 1) {
        unique = true
      }
    }
    if (!unique) {
      testSuite.remove(testCase);
    }
  }
}
```

Listing 4.9: Pseudo code for the removal of test cases which do not have unique coverage.

Listing 4.9 shows the pseudo code of our optimization approach.

Step 16: Write Results

The last two steps deal with persisting the result of the test generation (step 16). After the test generation and optimization is done (steps 6 to 14), we persist in this step the results. A typical implementation of this step, analogue to step 1, is to write the model, which may be in the computer's memory, from a storage, i.e. file or database storage. In our implementation of this step, we use a textual concrete notation for the model, and therefore serialize the model to that format.

as shown by (Heimdahl and George, 2004). However, if resources for testing are limited, the approach of removing test cases might be the only choice to execute the generated test cases against the SUT.

4.7. AZMUN: Implementation

In the previous sections, we explained our approaches for:

- creating test models using UML diagrams (Sec. 4.3),
- modeling test cases and their inputs and outputs (Sec. 4.4),
- selecting test cases using test selection criteria and test case specifications (Sec. 4.5),
- and finally how we generate test cases using a common workflow for test generation using model checkers (Sec. 4.6)

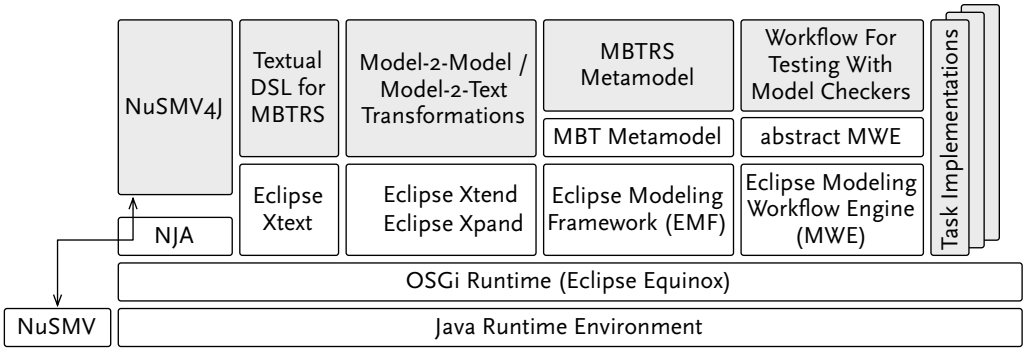


Figure 4.24.: Architectural overview of our software prototype AZMUN. The prototype extends our MBT metamodel and MBT abstract workflow system approach.

In this section, we briefly describe the technical details of our implementation which actually allows to import the UML based test models, graphical selection of test selection criteria, and execution of the test generation workflow by integration of the NuSMV model checker. Figure 4.24 shows the architectural overview of our implementation. We implemented our approach as an extension to our implementation of the MBT metamodel and the MBT abstract workflow system shown in Sec. 3.5 in the last chapter. The $MBTRS$ metamodel presented of this chapter (see Fig. 4.12) is modeled using *Ecore* and as an extension to the common metamodel $MBTMM$ (see Fig. 3.1). The common workflow for automated testing with model checkers is defined using the MWE syntax and can be found in Appendix F. The explained default task implementations for each step are realized using OSGi service registrations. Depending on the task, the implementation is done in pure Java, using the model-to-model transformation language *Xtend*, or using the model-to-text transformation language *Xpand*.

We decided to use a textual concrete syntax for the $MBTRS$ metamodel (called *Abstract Test Notation* (ATN)), and used the *Xtext* framework to create a rich editor, a parser and a pretty-printer for this language. Figure 4.25 shows the resulting editor, and additional extensions to Eclipse. To improve the usability of using our test generator, we also created a convenient Eclipse wizard which guides the user to select the UML components for test generation, and also the test selection criteria. Figure 4.26 shows screenshots of this wizard.

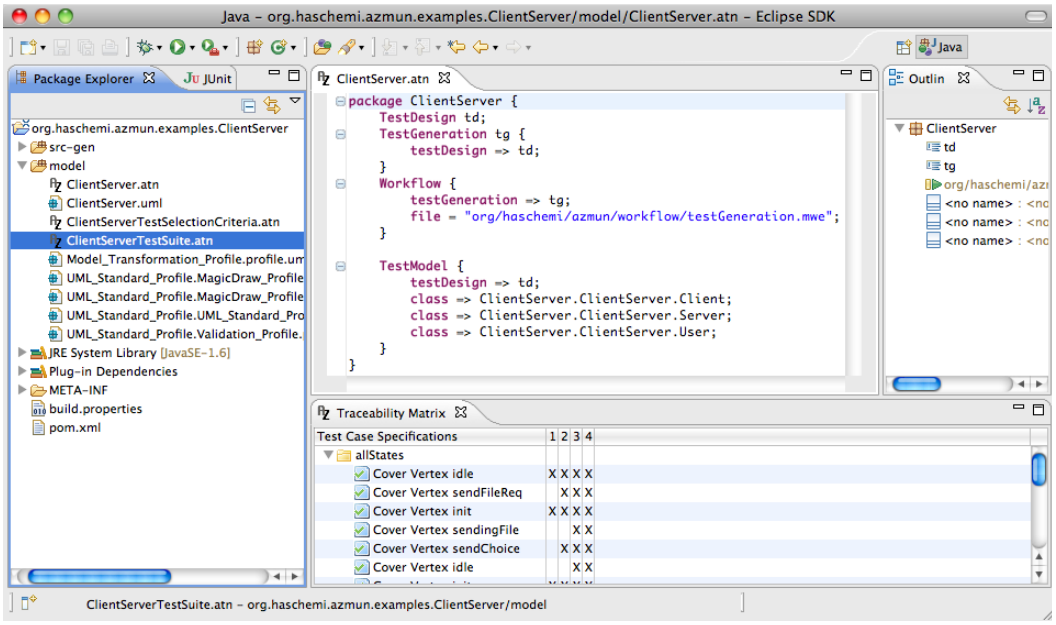


Figure 4.25.: Screenshot of Azmun's Eclipse integration. The generated editor for the ATN language is shown. Additional views like a traceability matrix show the contents of the ATN model in different ways.

4.8. Evaluation Of The Online Storage Example

In this section we report about the evaluation of our approach and developed tools using the Online Storage Example. In our setup we used the developed test model (Fig. 4.2 and Fig. 4.4) in our software prototype and automatically generated test cases using the presented test selection criteria 4.5. The primary goal of this evaluation is to show that our approach to fully automate the test generation using metamodels and workflow, as described in this and the last chapter, actually can be used to generate test cases for a non-trivial example. A secondary goal is to show how the test generation optimizations, which are provided

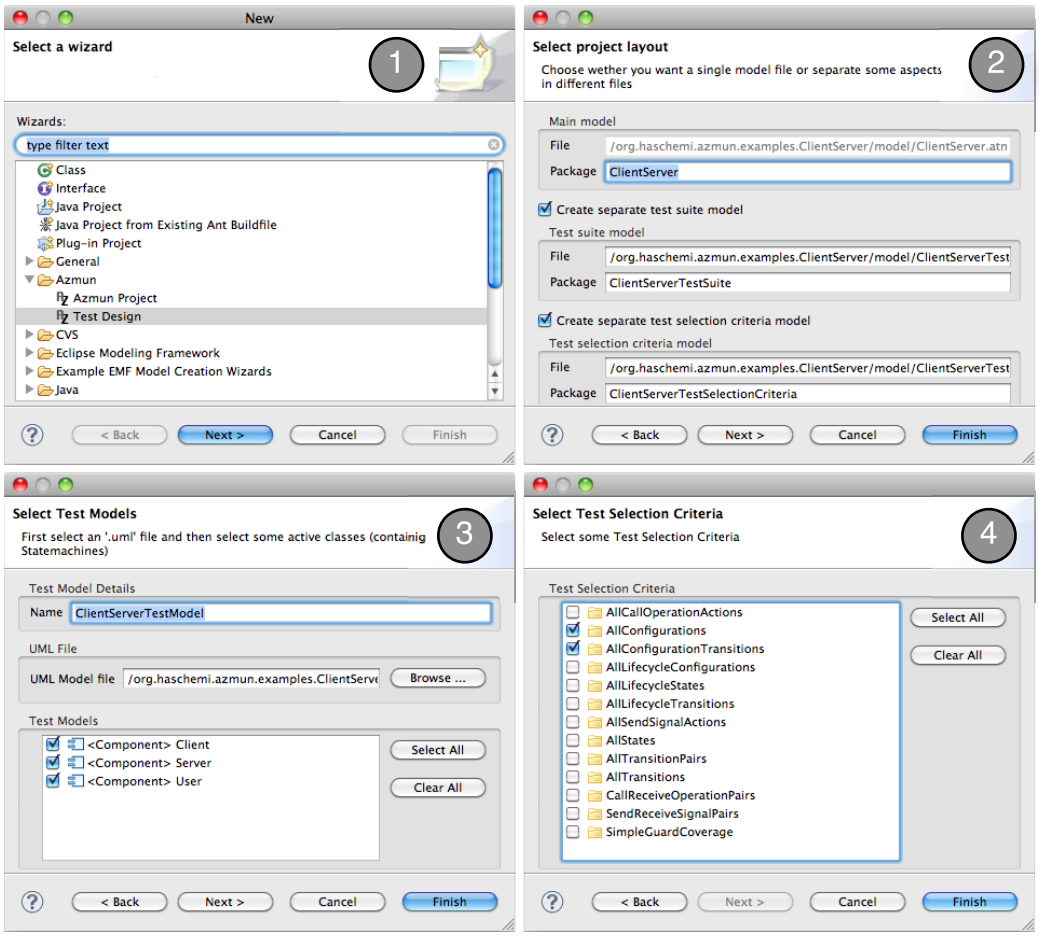


Figure 4.26.: Screenshot of Azmun's integration into Eclipse as a wizard to allow user-friendly creation of the needed configuration files. The wizards allows to select the components of the UML based test model, and selection of the test selection criteria.

as default implementations of the workflow steps, affect the test generation time, as well as the complexity of the generated test suite.

Assumptions and Setup

The complexity of the test suite is measured by the summary of all inputs and outputs in all test steps of all test cases. When executing test cases, both inputs and outputs require a communication to the SUT. The SUT will perform some actions (function calls, database access, network communication, etc.) based on the input of the test case. In addition, the SUT generates outputs so that a comparison between the expected and actual behavior of the SUT can be made. For this evaluation, we therefore assume that the sum of inputs and outputs correlates with the number of actions inside the SUT. The higher these values are, the more complex the execution of the test cases will be.

Test generation had been performed on an Apple MacBook Pro, 2.66 Ghz Intel Core2Duo CPU, 8 GB DDR3 RAM, and OSX 1.6. For each test execution, we measure the following values:

- **time:** The time (in seconds) the test generation took, measured as the difference between the start and the end time.
- **number of test case specifications:** The number of generated test case specifications as a result of the model transformation of the test selection criterion.
- **number of test cases:** The number of test cases generated by the workflow using the NuSMV model checker.
- **complexity:** The sum of all inputs and outputs of all test steps of all test cases.
- **coverage**: The coverage is a percentage measure between the total number of test case specifications and the test case specifications covered by all test cases.

The following parameters control the optimization of the test generation:

- **test suite minimization (true|false):** When enabled, the redundancy of the resulting test case is removed, which results in a smaller test suite size (Sec. 4.6).
- **monitoring (true|false):** When enabled, the generation of test cases for already covered test case specifications is avoided. This should lower the test generation time (Sec. 4.6).
- **AG only search (true|false):** This parameter is specific to NuSMV. If it is used, "a specialized algorithm to check AG formulas is used instead of the standard model

checking algorithms" (Cimatti et al., 1999). This parameter is only enabled for test selection criteria which are transformed to trap properties which only use AG formulas.

- **Cone of Influence (true|false):** This parameter is specific to NuSMV. When cone of influence reduction is active, the "problem encoded in the solving engine consists only of the relevant parts of the model for the property being checked. This can greatly help in reducing solving time and memory usage." (Cimatti et al., 1999)

Results

test selection criterion	test case specifications	test cases	complexity (input + output)	time (s)	coverage (%)
all-states	18	18	627	4	100.0
all-transitions	22	22	2003	10	100.0
all-configurations	23	17	823	5	73.9
all-configuration-transitions	60	39	3684	22	50.0

Table 4.1.: Test generation for the Online Storage Example using all presented test selection criteria. No optimization approach is applied.

The results of the unoptimized test generation are shown in Tab. 4.1. First, it is notable that we don't alway get a full coverage. The explanation of this effect is that for all-configuration and all-transitions, our heuristic of building a reachability tree (Sec. 4.5.3) produces configurations (combinations of vertices) which are infeasible. The model checker therefore finds no counter example for this combination. Another notable result is that generating test cases for transition-based test selection criteria takes more time and the resulting test suite is more complex, as for state-based criteria. This is an explicable effect since all-transition contains all paths through the state space as all-states. In scientific literature, this is also known as the *subsumption* relation (Clarke et al., 1985). A test suite that satisfies the subsuming test selection criterion also satisfies subsumed test selection criterion. The subsuming test selection criterion is considered stronger than the subsumed one (Weißleder, 2010).

Applying the optimization parameters for the test generation, we get the results shown in Tab. 4.2. Notably, the number of test cases and the complexity of all test suites is reduced. At the same time, the overall coverage is not changed. These results are explicable. The unoptimized results contain a lot of redundant test cases which do not increase the cov-

test selection criterion	test case specifications	test cases	complexity (input + output)	time (s)	coverage (%)
all-states	18	3	200	3	100.0
all-transitions	22	5	934	6	100.0
all-configurations	23	4	276	3	73.9
all-configuration-transitions	60	9	1317	20	50.0

Table 4.2.: Test generation for the Online Storage Example using all presented test selection criteria. Test generation optimizations were turned on.

erage of the whole test suite. These test cases are eliminated by the *test suite minimization* approach. The number of test suites and the total execution time is also improved by the *monitoring* approach since we avoid the generation of test cases in the first place. Although these results for the optimization look promising, it has been shown that the overall ability to detect faults is also reduced by this approach (Fraser, 2007).

However, for the focus of this dissertation we show that our approach and the developed tools can be successfully used for automated test generation. It should be kept in mind that the test cases were generated using only the default implementations of every workflow task. Much room is therefore left for optimization of parts of the test generation process. With the extensibility of the workflow, novel approaches can be integrated into the workflow.

4.9. Related Work

Several previous MBT approaches use UML to model the expected behavior of the SUT (Basanieri and Bertolino, 2000; Drusinsky, 2006; Nayak and Samanta, 2009; Weißleder, 2010; Peleska et al., 2011; Lasalle et al., 2011). They slightly differ in the supported UML diagrams and supported language constructs. While class diagrams are typically used for the structural (or data) part, the behavioral part is described using Statecharts, Activity Diagrams, Sequence Diagrams, etc. (Utting and Legeard, 2006). Our approach lines up with these tendencies by using UML class diagrams and Statecharts. However, as discussed before, the UML standard leaves some decisions open to the modeler, for example the language used to describe constraints, actions, and guards. Although OCL (Object Management Group, 2006b) exists, it is not intended to be used for action expressions. So every approach to use UML has to clarify the language and semantics of the used UML subset in detail. Existing approaches range from using general purpose languages like Java, C/C++,

or OCL like languages. In our approach, we defined an OCL like language, and its semantics are described using a transformation to an existing model checker language.

Using model checkers for automatic test generation has been proposed by several researchers (Kadono et al., 2009; Ammann et al., 1998; Gargantini and Heitmeyer, 1999). While these approaches show promising results, expressing the test model with the provided input languages of most model checkers can be too difficult for average test designers. To cope with this problem, graphical modeling notation like UML are transformed to the input model of model checkers (Lam, 2006; Kadono et al., 2009). This idea is also the core of our approach, where we transform our UML based test models to the NuSMV input model using model-transformations. The strength of our framework is that it is extensible by design, so changes to the transformation of UML to a model checker model can be improved or replaced. For example, it is possible to enhance the generation of trap properties for new test selection criteria, or even use a different model checker for test generation (Sec. 7.2).

In this chapter, we also showed by four examples on how test selection criteria can be formalized using metamodels and model-transformations. Several formalization attempts for test selection criteria have been proposed (Briones et al., 2006; Sadilek, 2010; Weißleder, 2010; Hong et al., 2001). In his PhD thesis, (Sadilek, 2010) proposes a formalization approach also based on metamodels and QVT for model-transformations. However, in his work the modeling goal is testing structural and behavioral aspects of metamodels, whereas our work focuses on testing reactive component systems. Another formalization approach for test selection criteria is presented by (Weißleder, 2010). This work defines a mathematical framework to describe the semantics of test selection criteria. The framework is then used to formalize transition-based, control-flow-based, and data-flow-based criteria. While this framework targets similar systems as our approach, the semantics of the test selection criteria are not defined in an executable form. In contrast, we used executable model-transformations to formalize test selection criteria.

4.10. Conclusion & Discussion

In this chapter we presented a formalization and a software prototype for the automatic test generation using model checkers for reactive component system. We contributed approaches for formalizing the artifacts and execution semantics of the test generation. Our contributions were a refinement of the common MBT metamodel with UML based test

models, test suites, test selection criteria, and test case specifications. A novel test selection criterion covering parallelism in Statecharts, called *All-Configuration-Transitions* ,was presented, and its semantics were shown. In addition, we showed how other typical test selection criteria, like *All-States* and *All-Transitions*, can be modeled using our approach. To round up the automatic test generation, we contributed a common workflow for testing with model checkers, which we used in our default implementation of the workflow steps together with the model checker NuSMV to generate test cases.

To achieve the above results, we first refined our formalization to a specific target system type (Sec. 4.2). To classify the system type, we used a taxonomy with seven dimensions where we positioned the systems we target. While this classification clearly shows what system we support, it also spots the general limits of our approach. For example, we choose *structural model coverage* and *test case specifications* for selecting interesting test cases. This decision leads us to the design of specific test selection criteria. Based on them, we defined model-to-model transformations to test case specifications, and finally to trap properties and verified counterexamples for these test case specifications. All of these conversions and operations depend on the initial decision in the taxonomy, and it is leaved unanswered if our formalization approach is applicable for different test selection methods like *data coverage* or *stochastic* selection. In addition to these limitations, we are also aware of limitations in the supported UML constructs. For example, we selected a subset of possible UML Statechart concepts, and leave concepts like *transition prioritization* and *history nodes* aside. The main reason for this limitation is the complexity of the transformation of the UML based test model to a semantically equal model checking problem. However, we think that our approach can be extended to support more test selection criteria and UML constructs in future with justifiable efforts.

CHAPTER 5

Model-Based Testing Of Dynamic Component Systems

5.1. Introduction

The last chapter served us as an intermediate step to show the hypothesis of this dissertation. We showed that using a common workflow for testing with model checkers, UML as the modeling notation for our test cases, and model transformations for the implementations of the workflow steps, we can systematically generate test cases for reactive component systems. In this chapter, we show how this approach can be applied to generate test cases for DCSs. The core of DCSs is the support of component configuration *evolution* during *runtime* of the system. Two characteristics allow this evolution, namely the *adaption of a component instance* and the *reconfiguration of the overall system* (Sec. 2.1). In these systems, component instances can withdraw provided functionality at any time, leading to the *dynamic availability* of functionality. An important aspect of dynamic availability is that it is not under the control of the component instance using the functionality. This requires that the component instances have to be prepared to respond at any time to arrival and/or departure of required functionality (Cervantes and Hall, 2003). This poses additional quality requirements to the development of software components, since missing or incorrect handling of dynamic availability can have an impact of the functionality of the overall system. In terms of software testing, any component code that is not prepared to handle dynamic availability correctly contains *errors* which might become *failures* of the overall system. Therefore, a systematic approach to test a component with focus on dynamic availability is desirable. The overall goal of this chapter is therefore to show how MBT can be used to systematically generate test cases for components with focus on dynamic availability.

The goals we aim to reach in this chapter are the following:

- Show how our formalization of MBT for reactive component systems can be adapted for automatic test generation of test cases for DCSs. The adaption should focus on testing the component's behavior to dynamic availability of functionality.
- The modeling notation for test models should be kept in the UML to avoid that test designers have to use yet another language.

To reach these goals, the following steps are taken in this chapter:

- We first show what the differences of test cases for common component systems and DCSs are (Sec. 5.2).
- Based on the identified characteristics of test cases for DCSs, we show how a UML profile can be used to enhance the test model to contain information about dynamic availability (Sec. 5.3).
- We then show how special test selection criteria can be derived from the extended test model. These test selection criteria guide the test generator in generating test cases which test the component's preparation to dynamic availability of functionality (Sec. 5.4).

5.2. Test Cases

Test cases for reactive component systems consist of a sequence of inputs to the SUT and some expected outputs to observe the behavior of the SUT. Based on our formalization (Chap. 3), inputs can be the change of a variable or the reception of a signal/operation. Outputs can be variable values, observed states, transitions, or sent signals (Sec. 4.4). With these primitives, it is possible to create arbitrarily complex test cases for reactive component systems. However, when the goal is to generate test cases to test the *dynamic availability of functionality* in DCSs, the test cases have to respect the component lifecycle (Sec. 2.1.2). The component lifecycle is an inherent state machine for all component instances. The component platform is responsible for actions like addition, replacement, or deletion of components. It drives this state machine by supporting platform actions like *install*, *start*, *stop*, or *uninstall*. In order to test a component's reaction to dynamic availability, the test cases have to move the component to the desired lifecycle state to observe its behavior in this situation.

For example, imagine two components A and B, where A depends on functionality pro-

vided by B. In order to test whether A correctly handles the withdrawal of B's functionality, we have to execute the following actions:

- Install the component A into the component platform.
- Install the component B into the component platform.
- Start the component B.
- Start the component A (A and B are wired, and the component instance CI_B starts providing its functionality to CI_A).
- Provide some inputs to CI_A to bring it to the desired state.
- Withdraw the provided functionality of CI_B.
- Observe CI_A for the expected behavior in case of a withdrawal.

This example shows that test cases for DCSs have to contain some behavior to respect the lifecycle of dynamic components. In our example, we had to install the two components, start them and interact with their component instances afterwards. Looking back to our primary goal to automatically generate test cases for DCSs, we finally have to change or enhance our test generation approach to generate such test cases. The next sections detail our approach to generate these test cases. It boils down to designing a special test model, and special test selection criteria.

5.3. Test Models

All MBT techniques have in common that they leverage on a formal model for systematic test generation. This test model is the primary source for the generation as it contains the oracle (expected behavior) information. In order to generate test cases for DCSs with the characteristics described in the last section, our approach is to include additional information about the component lifecycle in the test model. Adding this information is a twofold process: First, we add states and input signals to the test model (UML class diagrams and UML Statecharts). Second, we annotate these states and signals to be part of the component lifecycle by using UML stereotypes based on an UML profile.

We present our approach based on the Online Storage example we introduced in the last chapter (Fig. 4.4). For this chapter, we extend the example by a new feature where we allow the client to be *disconnected* from the server, a typical requirement in a client server architecture. For the client, this means that the provided service of the server (to upload a file) can become unavailable at any time. Thus, the client has to be prepared for this new require-

ment. Figure 5.1 shows the structural part of the extended test model. In this test model, we

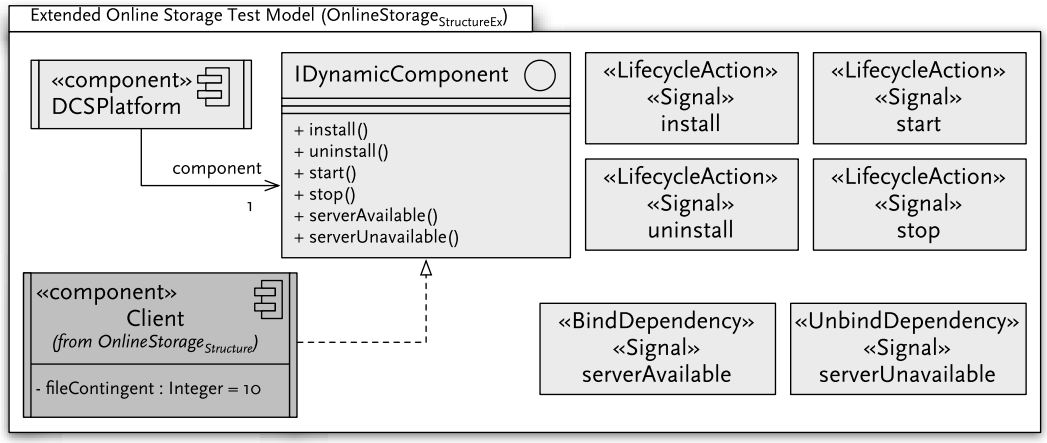


Figure 5.1.: Test model structure for the extended Online Storage Example. The Client component is extended with a new interface called IDynamicComponent, which adds signals for the dynamic component lifecycle. This figure refines the test model from Fig. 4.2.

extend the Client component with a new interface called IDynamicComponent. This interface represents the actions available in the lifecycle of a dynamic component. The first four actions install, uninstall, start, and stop, represent the corresponding actions of the lifecycle. Each of them is modeled as a UML signal, enabling the underlying state machine of component Client to react on these lifecycle signals. The last two signals, serverAvailable and serverUnavailable, represent the presence or withdrawal of the server functionality. When the connection to the server is lost, the serverUnavailable signal is fired by the platform. If the connection can be reestablished, the platform fires the serverAvailable signal. These two signals therefore notify the component instance about the availability of a service. All signals are annotated with a special *stereotype* based on the UML profile presented in Fig. 5.2. The first four signals are assigned with the stereotype LifecycleAction. This stereotype allows us later to distinguish between actions which change the component's lifecycle and normal actions belonging to the core functionality of the component. The last two signals are assigned with the stereotype BindDependency and UnbindDependency. These stereotypes allow us later to recognize signals which belong to the dynamic availability of functionality. The behavioral part of the Client component is shown in Fig. 5.3. The basic idea of this Statechart is to *embed* the expected behavior of the client in the component lifecycle. The original Statechart of the client component (Fig. 4.4) is present in the active state. We

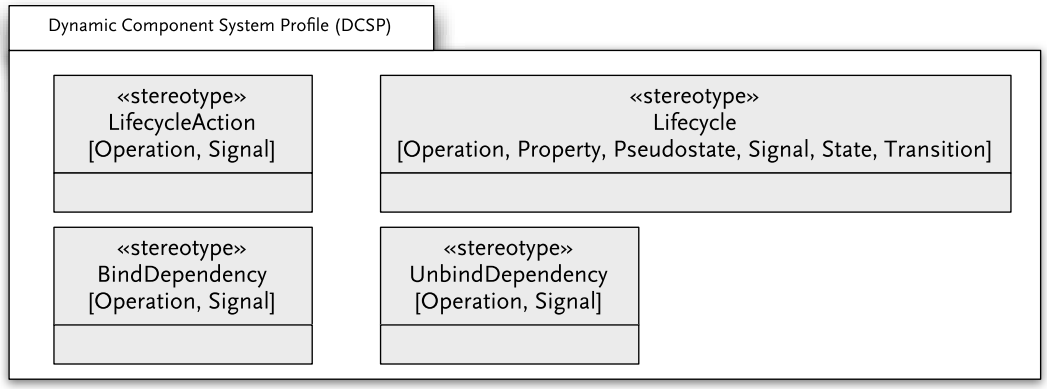


Figure 5.2.: A UML profile which defined stereotypes for annotating structural and behavioral elements of a test model.

also added more states to imitate the component lifecycle, namely the uninstalled, installed, and deactivated. The transitions between these lifecycle states are triggered by the actions of the IDynamicComponent interface. Like the signals in this interface, the lifecycle states are annotated with a stereotype based on the UML profile presented in Fig. 5.2. This stereotype allows us later to distinguish between states of the component's lifecycle and normal states belonging to the core behavior of the component. To model the dynamic availability of the server component, the installed state became an *orthogonal* state where the two regions of this state run in parallel. The lower region contains two states, serverAvailable and serverUnavailable, representing an active connection to the server or a disconnected state. The change between these two states is triggered by actions from the IDynamicComponent interface.

In the test model, we express our expectations on how the client should behave in the situation where the server is now reachable. We do this by storing the availability in a boolean variable called hasServer. We also changed the original transition t_1 and added a guard. This guard prevents sending a file to the server when it is not available.

The last fragment of the test model is the behavior of the DCS platform itself. From the perspective of the Client component, the DCS platform represents part of its environment. This environment triggers changes to the lifecycle of the component. The DCS platform itself has a defined behavior to respect the contract between platform and component instances. Figure 5.4 shows a simplified behavior of the DCS platform. This behavior is derived from the OSGi component lifecycle (Fig. 2.3).

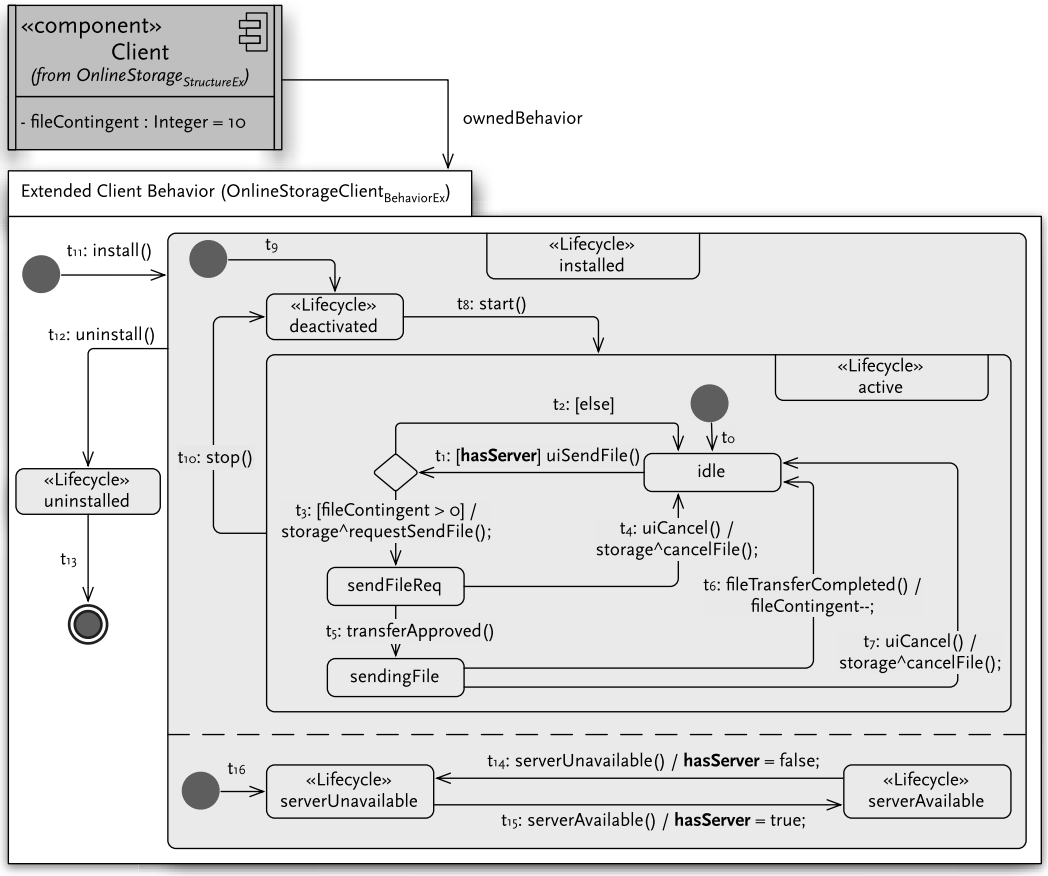


Figure 5.3.: Test model behavior for the extended Online Storage Example. The original Statechart (Fig. 4.4) is extended with states corresponding to the dynamic component lifecycle. Dynamic availability of required functionality is modeled in the orthogonal state installed in a parallel region.

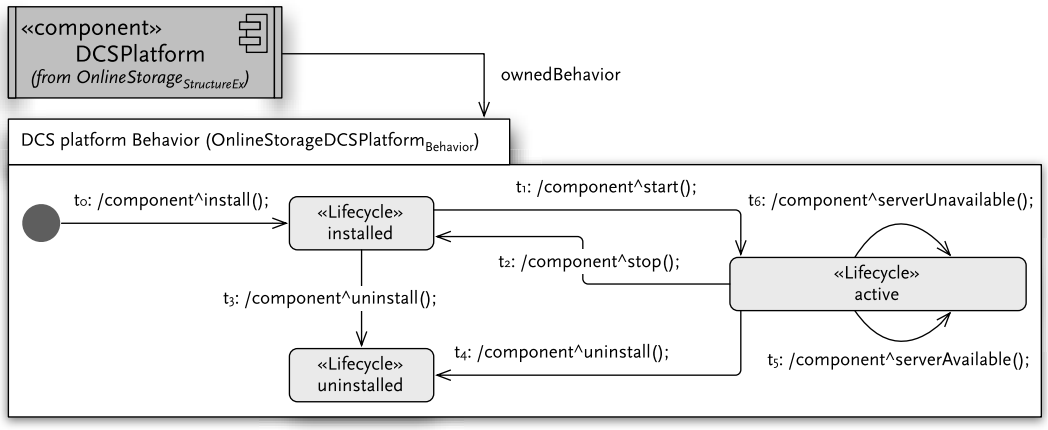


Figure 5.4.: Test model behavior which mimics the behavior of the dynamic component platform. The Statechart fires the lifecycle events install, uninstall, start, stop, and the events for signaling the dynamic availability of the Server component.

5.4. Test Selection

The test model approach we presented can be used as it is to generate test cases using our model checking based test generation. Because of the addition of states and signals, these test cases will contain the desired steps to test a DCS. To show this, we may use the test selection criterion *All-States* (Sec. 4.5.1) to generate test cases for our Online Storage example, which results in the following sequence of transitions: $t_{11}; t_9; t_{16}; t_{15}; t_8; t_0; t_1; t_3; t_5; t_{12}; t_{13}$. This corresponds to following actions in the SUT:

- the Client is installed ($t_{11}; t_9; t_{16}$)
- the server's functionality becomes available (t_{15}),
- the component is started (t_8),
- client is ready (t_0),
- since the server is available, the user can upload a file using the UI (t_1),
- the client (while having a sufficient file contingent) requests sending the file to the storage (t_3),
- the storage approves the file transfer (t_5),
- and finally the component is uninstalled ($t_{12}; t_{13}$)

Using general purpose test selection criteria (like All-States, All-Transitions etc.) allows us to generate test cases to judge about the the component's preparedness to dynamic availability. However, since resources are limited and we want to focus our test only on testing dynamic availability, we need special test selection criteria. In establishing these criteria, the specific states and signals introduced in our test models must be considered. Based on the previous presentation of possible test selection criteria for UML Statechart based test models in Sec. 4.5, we can derive component lifecycle specific selection criteria. They all have in common that they leverage on our UML profile (Fig. 5.2) used to annotate the test model.

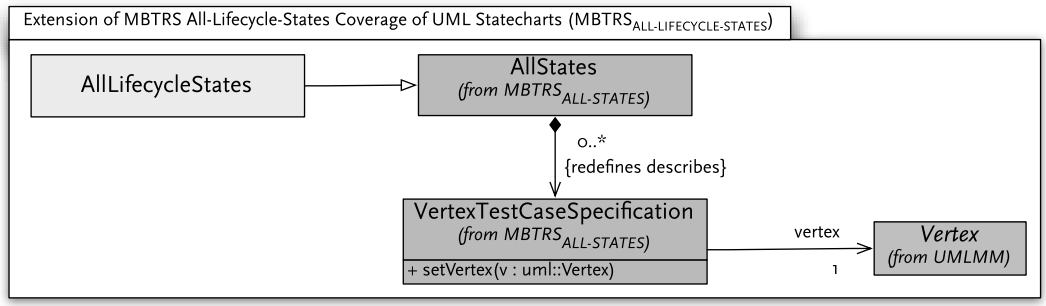
Definition 5.1 (all-lifecycle-states). All-Lifecycle-States coverage is achieved if every state of the model, assigned with the stereotype Lifecycle, is visited at least once.

Definition 5.2 (all-lifecycle-transitions). All-Lifecycle-Transitions coverage is achieved if every transition of the model, assigned with the stereotype Lifecycle, is visited at least once.

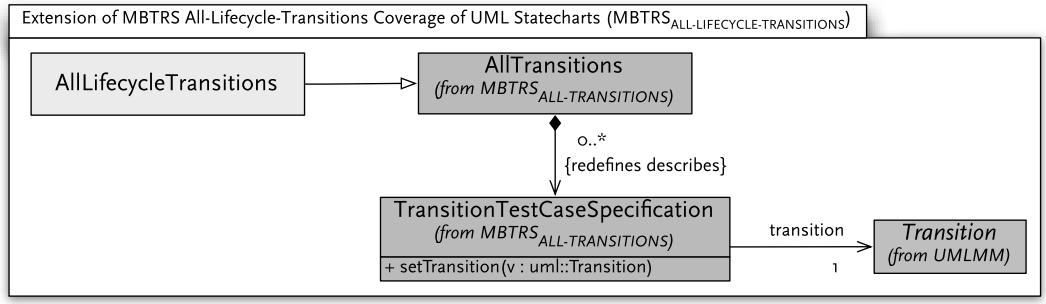
Definition 5.3 (all-lifecycle-configurations). All-Lifecycle-Configuration coverage is achieved if every configuration of the model, assigned with the stereotype Lifecycle, is visited at least once.

Fig. 5.5 shows the metamodels of the proposed test selection criteria. For the sake of brevity we discuss only the all-lifecycle-states criterion. The other criteria behave similar. Applying the definition for all-lifecycle-states (Def. 5.1) to our test models, full coverage is achieved when our test suite visits every vertex UML Statechart (simple state, pseudo state, combined state), assigned with the stereotype Lifecycle, at least once. For example, in our Online Storage Example (Fig. 5.3) the sequence of transitions $t_{11}; t_9; t_{16}; t_{15}; t_8; t_{12}; t_{13}$ gives *all-lifecycle-states* coverage of the Client component.

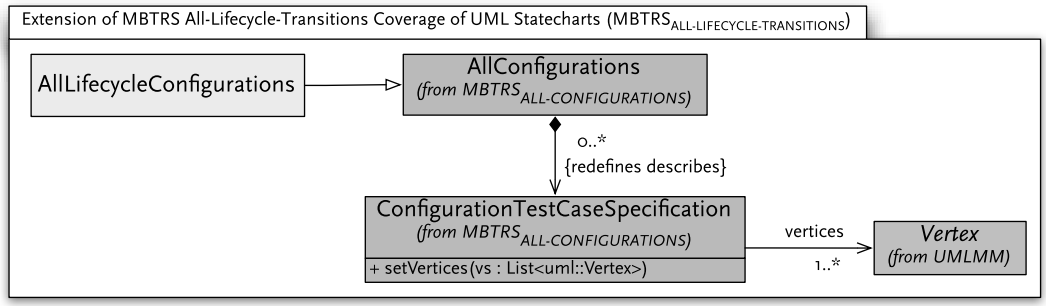
We model all-lifecycle-states as an extension to the all-states criterion (Fig. 4.13) in Fig. 5.5(a). We introduce the element AllLifecycleStates which extends the AllStates element of the $MBTRS_{ALL-STATES}$ model. This allows us to rely on the semantics of all-states coverage. The semantics of the all-lifecycle-states test selection criterion are therefore described similar to all-states (listing 5.1): We first collect all UML vertices (states and pseudo states) of all components (line 3). We then filter the vertex list by vertices with the stereotype Lifecycle. For each found vertex, we call the toTestCaseSpecification(Vertex) function. The called function creates a new metamodel element VertexTestCaseSpecification. Inside this function we operate in the context of the newly created metamodel element. So we can use the



(a) Metamodel for the All-Lifecycle-States test selection criterion. This figure extends Fig. 4.13



(b) Metamodel for the All-Lifecycle-Transitions test selection criterion. This figure extends Fig. 4.14.



(c) Metamodel for the All-Lifecycle-Configurations test selection criterion. This figure extends Fig. 4.16.

Figure 5.5.: Lifecycle-specific test selection criteria based on test models in Sec. 4.5.

```
1 toTestCaseSpecification(mbtrs::AllLifecycleStates als, mbtmm::TestDesign td):
2   allStates.describes.addAll(
3     td.testModelComponents().vertices()
4       .select(e|Lifecycle.isInstance(e))
5         .toTestCaseSpecification()
6   );
7
8 create VertexTestCaseSpecification toTestCaseSpecification(uml::Vertex v):
9   setVertex(v);
```

Listing 5.1: Semantics of all-lifecycle-states coverage. The listing shows a model-to-model transformation for describing the relation between the test selection criterion AllLifecycleStates and the model-specific test case specification VertexTestCaseSpecification. This transformation uses metamodel elements from $MBTMM$ (Fig. 3.1) and $MBTRS_{ALL-STATES}$ (Fig. 4.13).

declared methods to set the referenced vertex (line 9). The final step is to add every newly created VertexTestCaseSpecification element to the described reference (line 2).

5.5. Implementation

The described approach for automatic test generation for DCS has been implemented on basis of the Azmunframework (Sec. 4.7). We modeled the proposed UML profile in the UML tool *Magic Draw UML* (Fig. 5.6) and applied the stereotypes to the test model of the Online Storage Example. In order to process the new dynamic availability test selection criteria, we created a new implementation for the abstract task *(5) Compute Test Case Specifications* of the common workflow for automatic test generation with model checkers (Fig. 4.19). In addition, we extended the $MBTRS$ metamodel and added the test selection criteria metamodel elements. Fortunately, no other change was necessary in Azmun, which shows the flexibility of the abstract workflow approach taken for the process of test generation.

5.6. Evaluation Of The DCS-based Online Storage Example

In this section we report about the evaluation of the extended Online Storage Example using DCSs. In our setup we used the extended test model (Fig. 5.1 and Fig. 5.3) in our software prototype and automatically generated test cases using the the proposed test selection criteria which focus on dynamic availability. The primary goal of this evaluation is to show that our test selection criteria can effectively reduce the test suite size and the complexity of the

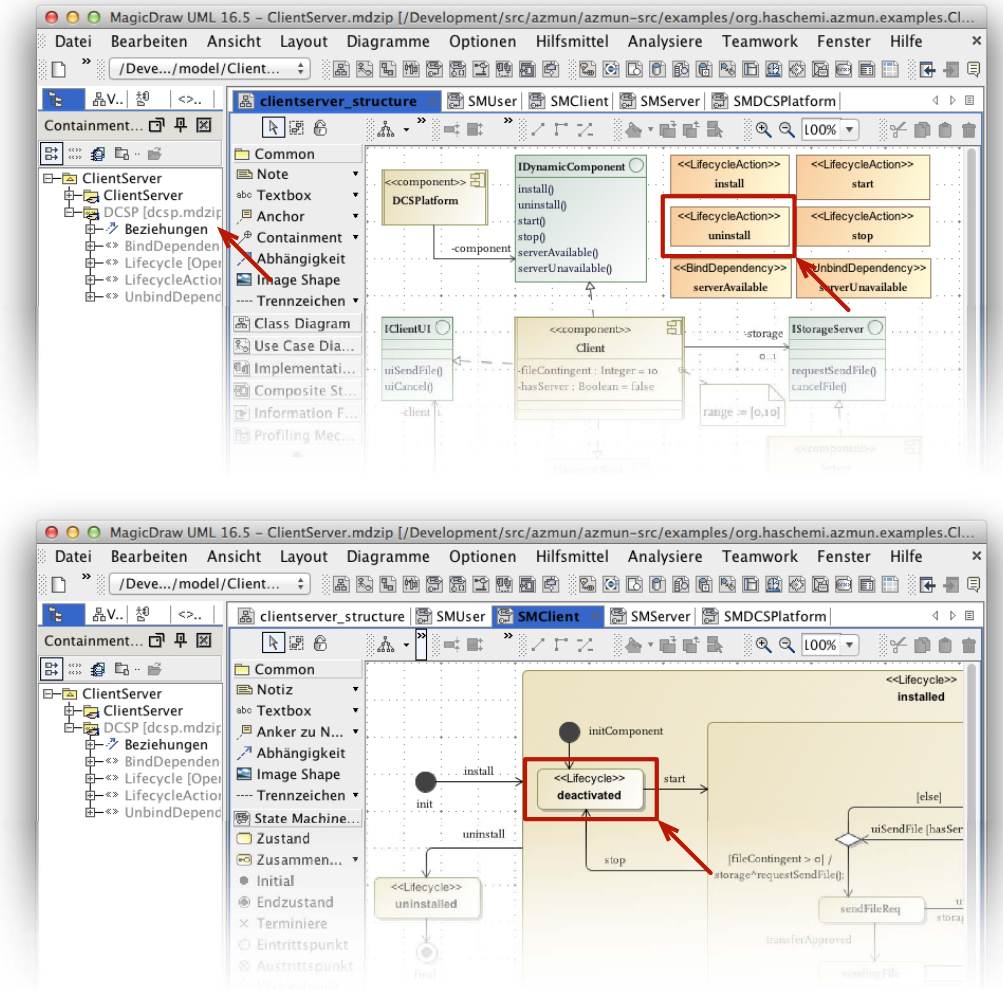


Figure 5.6.: Screenshot of the test model of the OnlineStorage Example in the UML tool *Magic Draw UML*. The highlighted areas show the application of the UML profile to the test model elements.

test execution, while testing a component's ability to react on dynamic availability.

Assumptions and Setup

For this evaluation, we use the same assumptions and setup like in the original evaluation of the Online Storage Example (Sec. 4.8). The only difference is that we put the test selection criteria all-lifecycle-states, all-lifecycle, and all-lifecycle-configurations into consideration, and test generation optimization is always turned on.

Results

The results of the test generation are shown in Tab. 5.1. Notable is the reduction of the number of test case specifications of the lifecycle-based test selection criteria compared to the classical ones. This is a direct result of the annotation of elements of the test model with stereotypes. This annotation can therefore also seen as a guidance to the test generation for a specific subset of the overall elements. The reduced size of the test case specification results in less test cases, with lower complexity. Since the lifecycle-based test selection criteria focus on dynamic availability behavior, we can be sure that the interesting parts of the test model are covered by the test suite. An interesting question at this point is which the fault detection capability the proposed test selection criteria have. Within this dissertation, we did not inspect this question and leave answers for it for future work (Sec. 7.3).

test selection criterion	test case specifications	test cases	complexity (input + output)	time (s)	coverage (%)
all-states	32	5	258	80	100.0
all-transitions	38	11	1853	87	100.0
all-configurations	46	11	919	81	69.6
all-configuration-transitions	148	39	5534	310	45.9
all-lifecycle-states	6	3	101	72	100.0
all-lifecycle-transitions	8	6	251	79	100.0
all-lifecycle-configurations	46	9	756	77	67.4

Table 5.1.: Test generation for the extended Online Storage Example using all presented test selection criteria. Test generation optimizations were turned on.

5.7. Related Work

Several approaches for model based testing of component systems have been proposed (Faivre et al., 2007; Kuliamin et al., 2003; Acharya et al., 2010; Broy et al., 2005). They mostly focus on generating tests for classical component models, where the component configuration is well known during development time and is only changed by stopping and starting the system. To support reusability of components, *built-in testing* has been developed as an alternative testing technique. For example, (Gross, 2004) describes an approach for using UML models to specify tests which are shipped together with the component. When assembling a system from components, the tests can be executed to improve the confidence that the configuration will work properly. The idea of built-in testing has been pushed further to *runtime testability* approaches, where tests are executed on a regular basis to monitor the current situation of the system (Gonzalez-Sanchez et al., 2010). However, the existing approaches to test component systems do not consider the generation of test cases which include the specifics of dynamic component systems, where functionality can become unavailable at any time.

5.8. Conclusion

In this chapter, we presented a systematic test generation approach for testing the component's preparedness to dynamic availability of functionality. Our approach relies on our previous formalizations of the structure and process of MBT. We first discussed what it is that makes test cases for DCSs special compared to test cases for traditional component models. In a nutshell, we had to include special actions related to the component platform into the test cases. To be able to generate such test cases we presented the following approach: First, we extended the component's interface to be able to receive lifecycle signals like installed or started, or wiring-specific signals like serverAvailable. Second, we explicitly modeled the component lifecycle in the UML Statechart by introducing new states like installed or activated. Both the structural and the behavioral part of our test model leverage on a UML profile containing stereotypes for the lifecycle states, transition, signals, etc. We annotated our test model with these stereotypes to be able to distinguish the lifecycle elements from the elements related to the normal test model. We also modeled a new component which represents the behavior of the DCS component platform. The UML Statechart of this new component implements the component lifecycle and sends the lifecycle signals to the

component under test. It therefore acts as the environment for the component under test. Finally, we proposed structural test selection criteria focusing on testing the component's preparedness to dynamic availability. The main idea for all proposed test selection criteria is to filter the test model elements based on the UML profile stereotypes. We implemented our approach on basis of the Azmun framework, the MBT testing tool implemented in this dissertation.

CHAPTER 6

Case Study: SOSEWIN Alarm Protocol

In this chapter we report about a case study for which we used the developed approaches and tools of this dissertation. The goal of the case study was to test the software of a self-organizing distributed earthquake early warning system with focus on dynamic availability. The software of this system is developed using model-based tools and languages like SDL-RT, UML, C++, and ASN.1. Although the system is not built with principles of DCSs as defined in this dissertation, its support for self-organizing networks, where network nodes may fail at any time, and representation of these failures inside the software, allowed us to interpret it as a DCS. For this case study, we chose one software component of the earthquake alarm protocol which had to resist dynamic availability of provided functionality by another component. We created a test model based on the SDL-RT specification using our UML based formalism, and used our software prototype to automatically generate test cases for this component.

In the next section, we give a short overview of the self-organizing seismic early warning system (Sec. 6.1). Next, we present in Sec. 6.2 a test model focusing of one component of the earthquake alarm protocol. In Sec. 6.3 we present our results for automatic test generation using our software prototype Azmun.

6.1. Introduction

Compared to current earthquake early earning systems, which use expensive and highly sensitive sensor stations connected to a single data center, the system of our case study follows a novel approach by relying on Self-organizing wireless mesh networks (WMNs) (Fis-

cher et al., 2012). WMNs consist of nodes that communicate wirelessly. Without a central administration, the nodes establish a network topology for multi-hop communication. Nodes can leave and join the network at any time: Existing nodes may loose connection, may be destroyed, or may run out of energy; new nodes may be inserted or nodes may recover their lost connection. Therefore, the topology of the network fluctuates, and global knowledge about all nodes in a WMN is missing (Akyildiz et al., 2005). Such a WMN is the technical platform of the self-organizing Seismic Early Warning Information Network (SOSEWIN) (Zschau et al., 2007), a project of the European Union our working group is involved in. The goal of SOSEWIN is to develop a distributed earthquake early warning and disaster management system. This system consists of nodes that form a WMN. The nodes are deployed in a city and they are equipped with acceleration sensors. In case of an earthquake, the nodes sense non-destructive seismic waves that arrive at the city before destructive waves arrive. The nodes compare their measurements and, communicating via the WMN, vote whether an early warning should be triggered. Furthermore, the nodes record all seismic waves. From this data, a shake map can be generated, which can tell disaster relief workers where they can expect the severest damages. Additionally, the WMN may serve as a backup communication medium if traditional communication systems get destroyed by the earthquake.

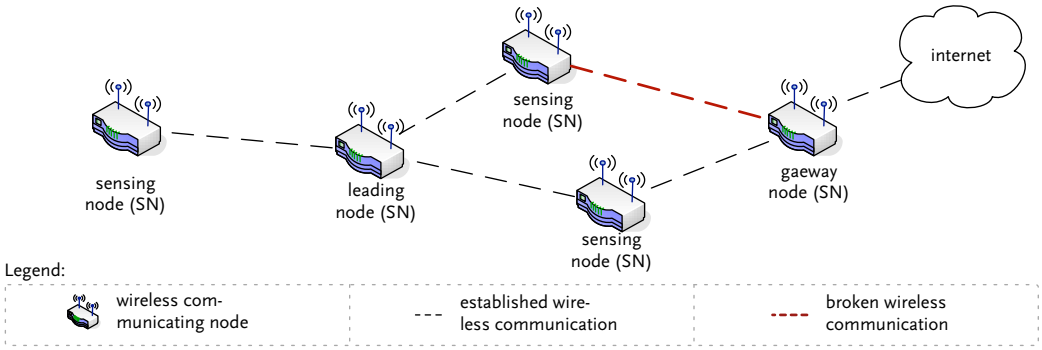


Figure 6.1.: SOSEWIN network composed of sensing nodes (SNs), leading nodes (LNs), and gateway nodes (GNs). SOSEWIN is a self-organizing networks where disconnection and failure of nodes are handled by novel routing mechanisms.

A typical SOSEWIN consist of the following nodes types (Fischer et al., 2009), as illustrated in Fig. 6.1:

- *Sensing node (SN):* These nodes monitor the ground shaking.

- *Leading nodes (LN):* These nodes are SNs with a special *leading* property derived from the clustering scheme of the network.
- *Gateway nodes (GN):* These nodes are SNs that act as information sinks and connection to end users outside the network.

While SNs, LNs, and GNs fulfill different tasks, their hardware and software is identical. This is a crucial prerequisite to support self-organization. For example, a destroyed *leading* node may result in a clustered network where some part of the network is not reachable. To recover this area, another SN has to be chosen to be the leader for this cluster.

The software of SOSEWIN nodes is described using SDL-RT models. SDL-RT is an object oriented, graphical language based on SDL. It extends SDL with real time concepts and allows usage of modern C types. In version 2.0 of the standard, the support of UML diagrams has been added to support organization of classes and representation of the physical architecture, and how different nodes communicate with each other (Specification and description language - real time (SDL), 2013). In Fig. 6.2, the basic building blocks of the SDL-RT specification and the exchanged messages of the alarm protocol are shown. Each block's behavior is further detailed in other SDL-RT diagrams as exemplary shown in Fig. 6.3. Using these models, several transformations are available to transform the models to C++ code. The target of this code can be either a network simulator, or the real devices. To support the development of these systems, several tools and frameworks exist. The most important ones are

- a model repository for centralized management of models,
- an experiment management system that supports execution of experiments and storage of their results,
- and GIS-based technologies to visualize or administrate the network.

For the sake of brevity, we will not go into further details of SOSEWIN's architecture and refer to published results (Fischer et al., 2009, 2012; Ahrens et al., 2009).

6.2. Creating The Test Model

The alarm protocol is an application created on basis of the SOSEWIN platform. For our case study, we first had to answer if this application really can be seen as a DCS, although it does not use a dynamic component platform in its technical implementation. It turned out that the principles of DCSs were present in the architectural level, but also in the SDL-RT

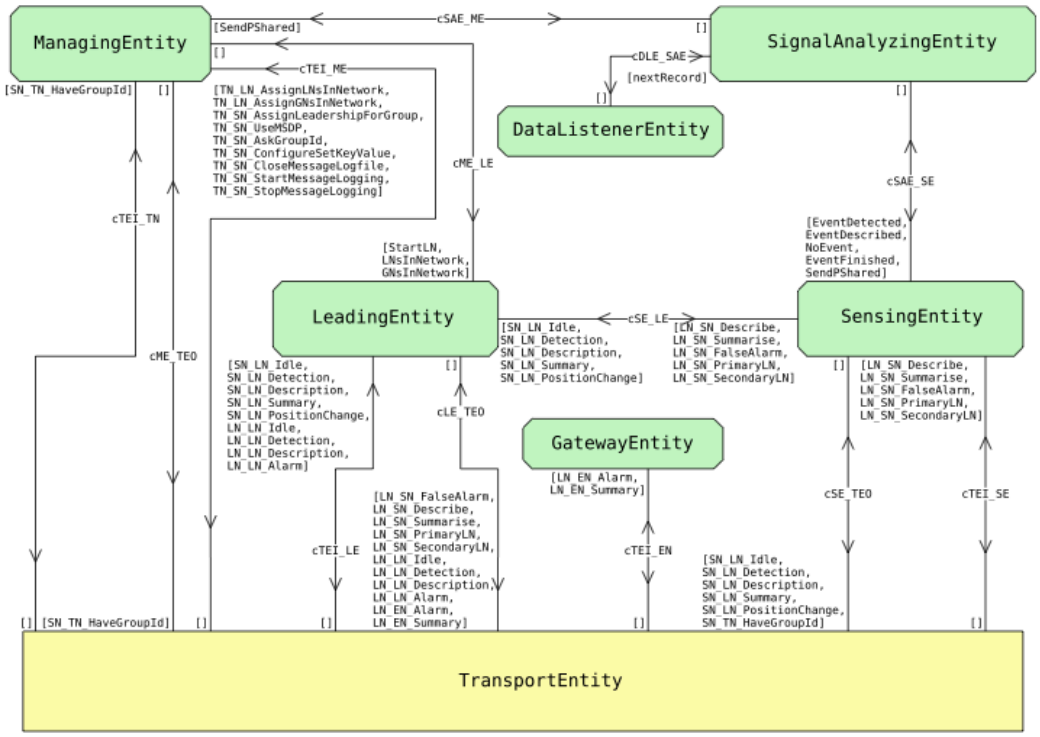


Figure 6.2.: SDL-RT specification of the alarm protocol. Image by (Fischer et al., 2012).

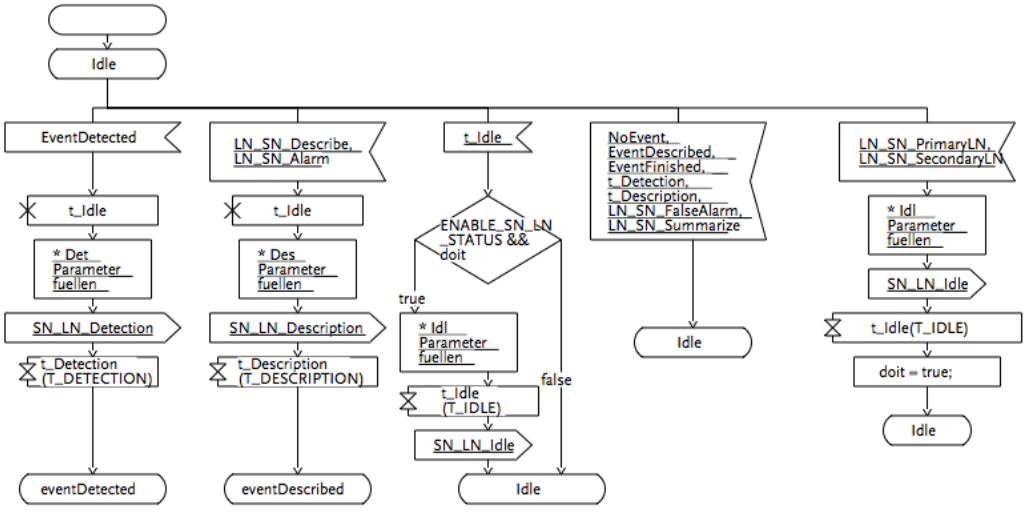


Figure 6.3.: SDL-RT State Machine of a Sensing Entity. Image by (Fischer et al., 2012).

models. Fig. 6.4 shows our view on the SOSEWIN as if it's software were developed using

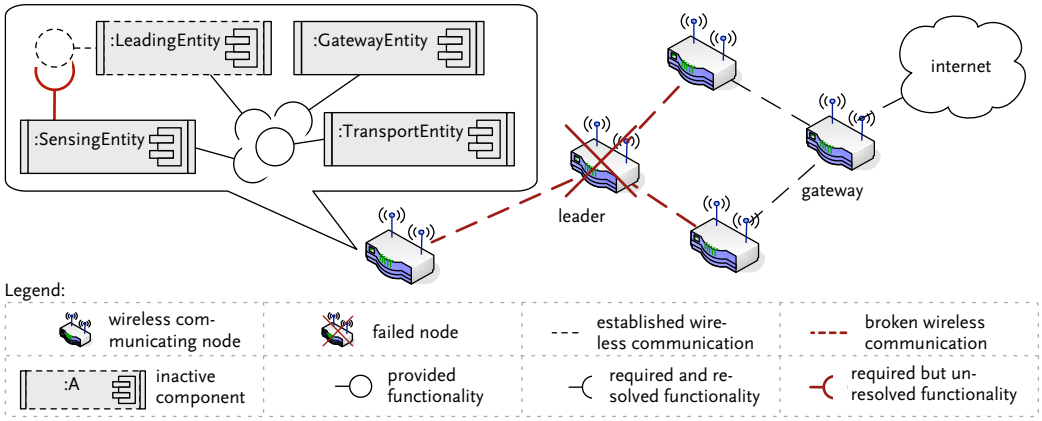


Figure 6.4.: SOSEWIN with a destroyed LN, and representation of this failure as a deactivation of a DCS component instance.

a DCS. Each node in the system would contain a dynamic component platform, where component instances provide their functionality to other component instances. Some component instances would withdraw their provided functionality based on the state of the node or network conditions. For example, the component instance LeadingEntity would represent an active connection to the *leader* of a network cluster. When the network connection is be available and stable, the component instance would provide its functionality to communicate with the leading entity. However, communicating with the component instance actually would result in a real network communication over the wire. If for some reason the communication to the leading node breaks (Fig. 6.4), this lack of communication would be represented as a withdrawal of the functionality of the LeadingEntity component instance.

Any component using LeadingEntity's functionality has to be prepared for the dynamic availability of this functionality. In SOSEWIN, the SensingEntity is one of the central components. It senses seismic waves, performs algorithms on the continuous signal stream, and alarms its leader if an earthquake is guessed. For this case study, we therefore selected the SensingEntity as the *component under test*. Based on the SDL-RT model of the sensing entity, we created a UML based test model. The structural and behavioral aspects of the test model are shown in Fig. 6.5, 6.6, and 6.7. In order to keep the test model small and focused, we decided to abstract some of the details of the original SDL-RT model out. First, we only modeled the behavior of the SensingEntity in detail, since we were interested in

its reaction to dynamic availability. However, to meet the requirements of the SensingEntity, other components still had to be created to mimic the behavior of the environment. Another abstraction was to only include the needed signals for the communication of the LeadingEntity and the SensingEntity, but skip any other signals for the communication with other SDL-RT blocks. Finally, we skipped the rich support of SDL-RT to trigger timers. Instead, we assumed that some component representing the environment will trigger the timer events when necessary (Fig. 6.7).

In the version of the SDL-RT specification of the alarm protocol used in this case study, the SensingEntity was only partly prepared for dynamic availability of the LeadingEntity. While the presence of a new leader was considered using an event called LN_SN_PrimaryLN, no event existed that notified the SensingEntity about the *absence* of the leader. We therefore designed a new event called PrimaryLN_Lost for this situation. Although the system did not support this behavior yet, the MBT approach allowed us to anticipate the needed changes of the SUT, because the test model and the SUT are not tied to each other. We modeled the expected behavior of the sensing entity in a UML Statechart in Fig. 6.6. We followed our approach to explicitly model the lifecycle states and transitions. We further modeled the presence/absence of the leader functionality using an orthogonal region, so that dynamic availability is supported at any time, independent from the core functionality in the upper region of the Statechart. To be able to generate test cases for this model, we had to create additional Statecharts to emit events which drive the component under test (Fig. 6.7). These Statecharts are designed to emit events non-deterministically, so that any of these events can be triggered at any time. However for the DCSPlatform component, we followed our modeling approach and added details about the lifecycle of the DCS component, and triggering of the presence/absence events of the leading entity.

6.3. Evaluation

Using the presented test model, we used our test generation approach and tool to generate test cases for the alarm protocol. The setup of the test generation was equal to the setup of the Online Storage Example (Sec. 4.8). The results of the test generation are shown in Tab. 6.1.

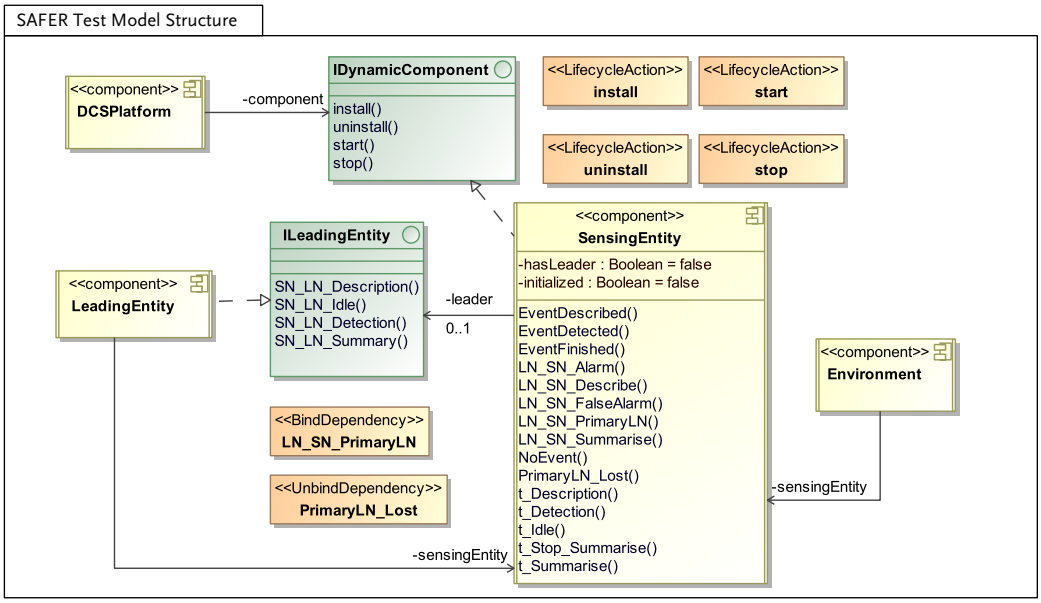


Figure 6.5.: Test model structure for the Sensing Entity component of the alarm protocol. The DCSPlatform component sends events to trigger a change of the lifecycle status of the SensingEntity component. The Environment component represent an environment firing timer events. The LeadingEntity non-deterministically sends events of the alarm protocol to the SensingEntity.

test selection criterion	test case specifications	test cases	complexity (input + output)	time (s)	coverage (%)
all-states	30	5	262	44	100.0
all-transitions	62	24	1355	46	100.0
all-configurations	113	26	1464	56	89.4
all-configuration-transitions	232	58	3564	486	43.1
all-lifecycle-states	6	2	64	53	100.0
all-lifecycle-transitions	8	3	100	37	100.0
all-lifecycle-configurations	51	16	876	42	74.5

Table 6.1.: Test generation for the test model of the alarm protocol's sensing entity. Test generation optimizations were turned on.

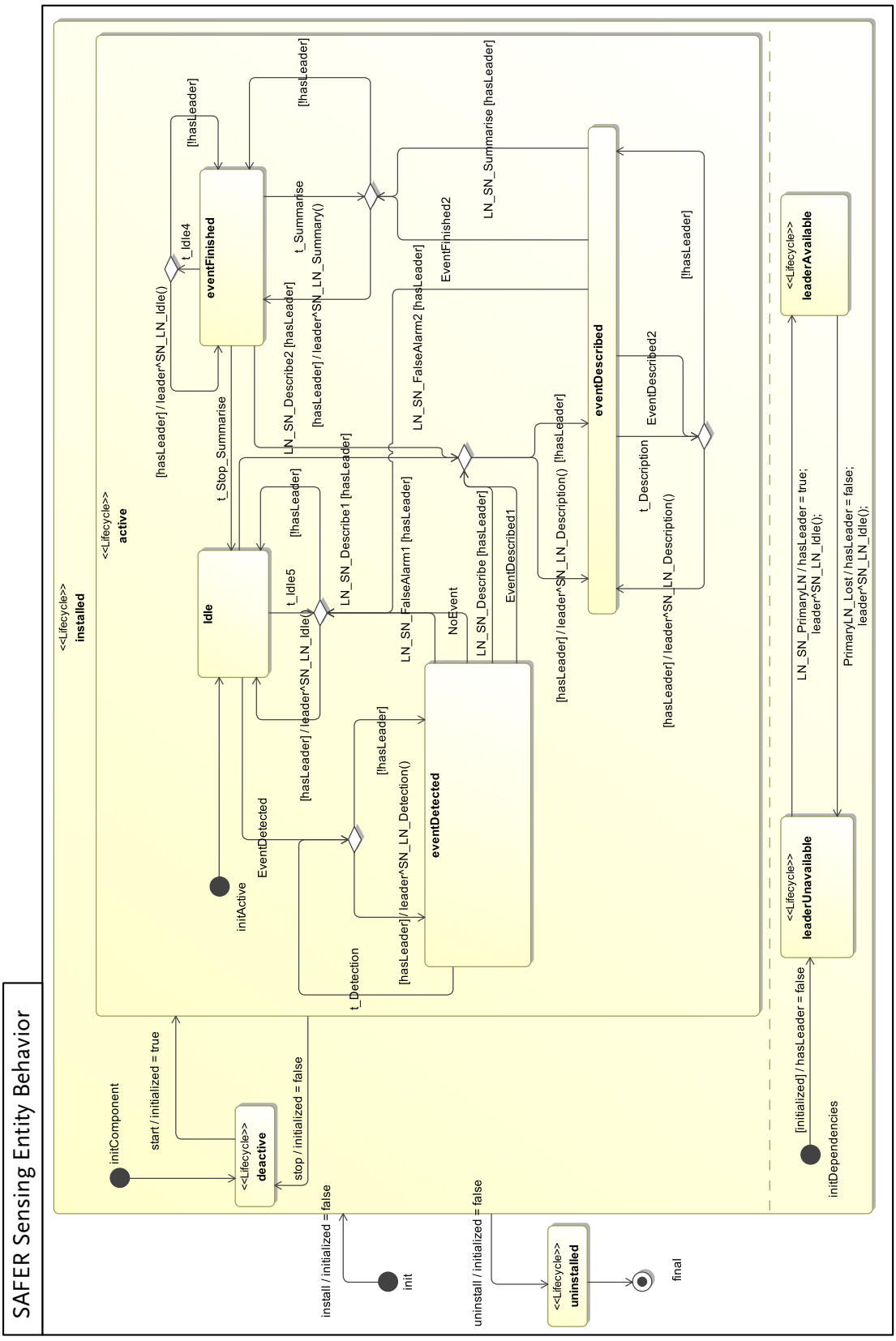


Figure 6.6.: Test model of the Sensing Entity component of the alarm protocol.

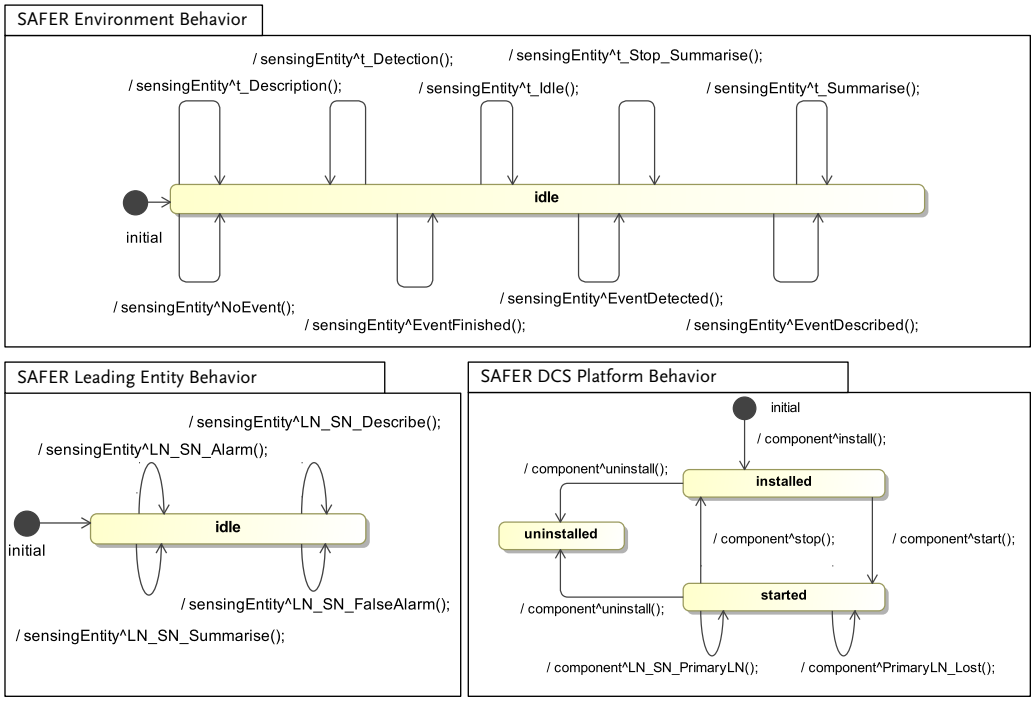


Figure 6.7.: Behavior models of the components Environment (top), LeadingEntity (lower left), and DCSPlatform (lower right).

CHAPTER 7

Conclusions

Within the structure of this dissertation, each chapter is concluded by itself. In this chapter, we therefore give an overall conclusion of the dissertation. In Sec. 7.1, we discuss whether the hypothesis of the dissertation has been shown. The impact of our work is then shown in Sec. 7.2. Finally, we discuss future work in Sec. 7.3.

7.1. Hypothesis And Aim

The aim of this dissertation was to answer whether this hypothesis holds:

> *Model based testing can be applied to systematically test Dynamic Component Systems.*

In order to answer this question, we made several contributions as shown in Sec. 1.5. We first formalized the artifacts and processes of MBT in a novel way, in which we combine metamodel and workflow techniques in order to be able to automate the test generation. This step served us as basis for the formalization's next refinement. In the refinement process, we decided to focus on reactive component systems and chose UML diagrams to create test models for these systems. We were able to model an abstract workflow for testing with model checkers, which finally allowed the complete automation of the test generation process. With this refinement process, we showed that MBT can be applied to generate test cases for classical component systems. Although this has already been shown in existing literature, it serves us as an intermediate step to show the hypothesis of this dissertation. To actually show that the hypothesis holds, we first had to analyze what specific requirements

test cases for DCSs have, and how they differentiate to test cases for classical component systems. It turns out that the addition of signals corresponding to events in the dynamic component platform would allow us to reach the goal. In order to reuse the previous approach of generating test cases for classical component systems, we therefore proposed an UML profile together with a convention for the UML test models to integrate the expected behavior of the DCS inside the test model. This finally allowed us to generate the needed test cases. We therefore conclude that the hypothesis of this dissertation has been shown.

7.2. Impact

In this thesis we chose a formalization approach for MBT which allowed us to directly implement our ideas based on existing Eclipse and OSGi technologies. Apart from that, we made the process of test generation explicit using workflow technologies. Although we contributed a detailed workflow for testing with model checkers and provided implementations for *every* step in that workflow, our approach allows to exchange every step in this workflow with an own implementation. The main idea was to motivate scientists to leverage on this framework (Azmun) to experiment with new test generation approaches, or simply use the framework to generate test cases for their application. Through the time of working on this dissertation, a number of interesting projects that are directly based on the ideas and technologies in this dissertation have been published:

- *NuSeen* (Arcaini et al., 2011, 2013b), a project of the *Formal Methods and Software Engineering Laboratory* of the university of Bergamo, is a *NuSMV model advisor* which tackles the problem of automatic reviewing NuSMV formal specifications. It provides an extensive list of logical checks which exhibits vulnerabilities and defects in NuSMV models. In this dissertation, in order to interact nicely with the model checker during test generation, we developed a metamodel and an EBNF grammar for the NuSMV input model. This gave us a rich Eclipse editor for the NuSMV input model language, as well as a parser and pretty printer. The NuSMV model advisor leverages on our parser to build the abstract syntax tree of a model to perform its checks.
- The *Real-Time System Lab* of the Technical University of Darmstadt used Azmun for their research into *Software Product Line Testing* (Cichos et al., 2012, 2011; Cichos and Heinze, 2011). They used so called 150% models to derive small test suites for single products of a product line. Their work represents the biggest modification to the

default implementation of Azmun so far. With our flexible formalization approach, *no change* to the core functionality of our framework was needed. For the sake of brevity, we briefly present two interesting use cases:

- First the NuSMV model checker was replaced with the Spin model checker because of performance issues. In Azmun, this can be done easily by implementing three abstract tasks for *input model generation*, *trap property generation* and the actual *verification* which involves communication with the model checker.
- The second modification made by this group was to change some parts of the common workflow for test suite reduction (Cichos et al., 2011). The basic idea was to combine test case specifications in order to generate longer test cases with greater coverage than test cases with uncombined test case specifications. In order to implement this idea, a refinement of the test generation step of Azmun was necessary.

- Some students use our approach and tools for their theses:
  - Harald Chichos used Azmun in his PhD thesis to implement a novel test suite reduction approach and generating test cases for product lines (Cichos et al., 2011; Cichos, 2013).
  - Oksana Kolach used Azmun in her diploma thesis to implement Mutation Analysis (Kolach, 2011). She implemented several alternative workflow steps where she generated mutants of the UML test model. Then she executed the original test suite against the mutated model to measure the quality of the test suite.
  - Martin Thomas used Azmun in his bachelor thesis for testing web applications (Thomas, 2011). He implemented a model-to-text transformation which translates our MBT metamodel based test cases to JUnit test cases using the Google Selenium framework. With this, he was able to generate a test suite to test scenarios for input forms of a big industrial use case.

7.3. Future Work

As with any dissertation, many topics remain open which have to be detailed in future work. At the same time, our approaches and technologies bring up new research questions. In this section, we discuss which work may follow after this dissertation.

Formalization of MBT

Apply formalization to different modeling notations and target system types. Up to this point, it is unclear if our formalization approach for MBT using metamodels, abstract workflows and model-transformations can be applied to more modeling notations and target system types. In this dissertation we focused on UML and (dynamic) reactive component systems. However, as we identified in previous discussions, we may have done design decisions which might have resulted in a metamodel/workflow which serves our purposes only. Future work is therefore needed to show the applicability to other domains.

Reproducibility Of Test Generation. Our formalization using abstract workflow allows the replacement of every workflow task through the *task mapping* technique. This was an important design decision to support future MBT technologies and research ideas. However, it reveals the question of reproducibility of the test generation results.

Test Generation for DCSs

Evaluate The Effectiveness Of Testing. As part of our formalization, we proposed the test selection criteria *All-Configurations* and *All-Configuration-Transitions* which focusing on parallelism in UML Statecharts (Chap. 4). We also proposed some test selection criteria focusing on test generation for DCSs (Chap. 5). The goal of testing is to reveal errors on the SUT by trying to find faults during the execution of the system with exemplary input. The capability to reveal errors is therefore an important measure for the quality of the generated test suite. However, the contents of the test suite are primarily influenced by the test selection criteria used for the test generation. The quality of selection criteria can be expressed with their *fault-detection capability*. One way to judge the fault-detection capability of the proposed test selection criteria is to do a *mutation analysis*. In this method, a set of *mutants* that are representative of realistic faults are created. If a generated test case can distinguish between the mutant and the original program, the mutant is *killed*. The success of how many mutants a given set of test cases can kill is measured in the *mutation score*, where a high mutation score means a high fault-detection capability. In order to gain more information about the quality of the proposed test selection criteria, future work should investigate the measurement of the mutation scores for the proposed criteria.

Reduce Modeling Effort. Our approach to generate test cases for DCSs is to rely on a convention for test models. This convention requires that the component lifecycle is modeled explicitly in the UML class diagram and Statechart. While this is a sensible approach to show that our approach can be used with existing UML tools, it burdens the test designer with additional work. An interesting extension for future work would be to (partly) automate the modification of existing test models with DCS aspects. This could be done by enhancing existing UML tools, or applying model-transformations in a pre-processing step of the test generation workflow. Although this idea would reduce the modeling effort, it might be difficult to implement because dynamic availability is not a cross-cutting concern. More research is therefore needed to analyze under which circumstances the modeling effort of creating test models for DCS can be reduced.

APPENDIX A

Grammar for Expressions used in UML Diagrams

Range Expression	Predicate Equivalent
[0,4)	$0 <= x < 4$
[1,6)	$1 <= x <= 6$
(3,5)	$3 < x < 5$
(2,4]	$2 < x <= 4$
3	$3 <= x <= 3$

Table A.1.: Some examples of version range strings and their predicate equivalent. Note that a simple range (e.g. *4*) indicates a range which is exactly the specified integer.

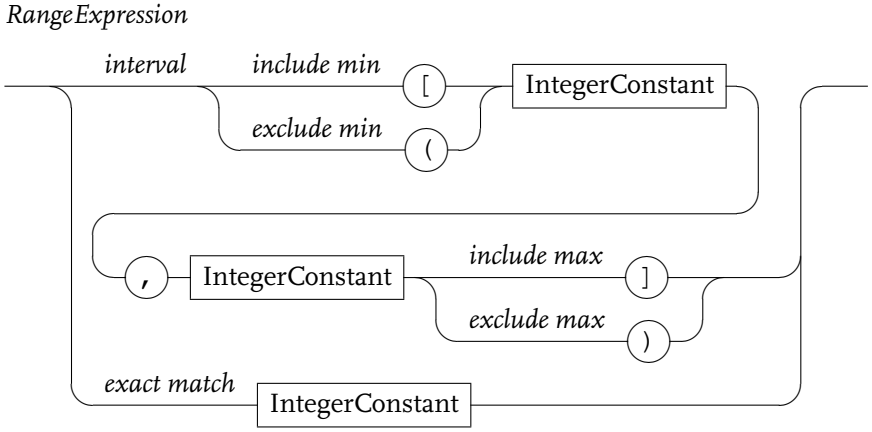


Figure A.1.: Rule for range expressions used in UML class diagrams to limit the value range of integer-typed attributes

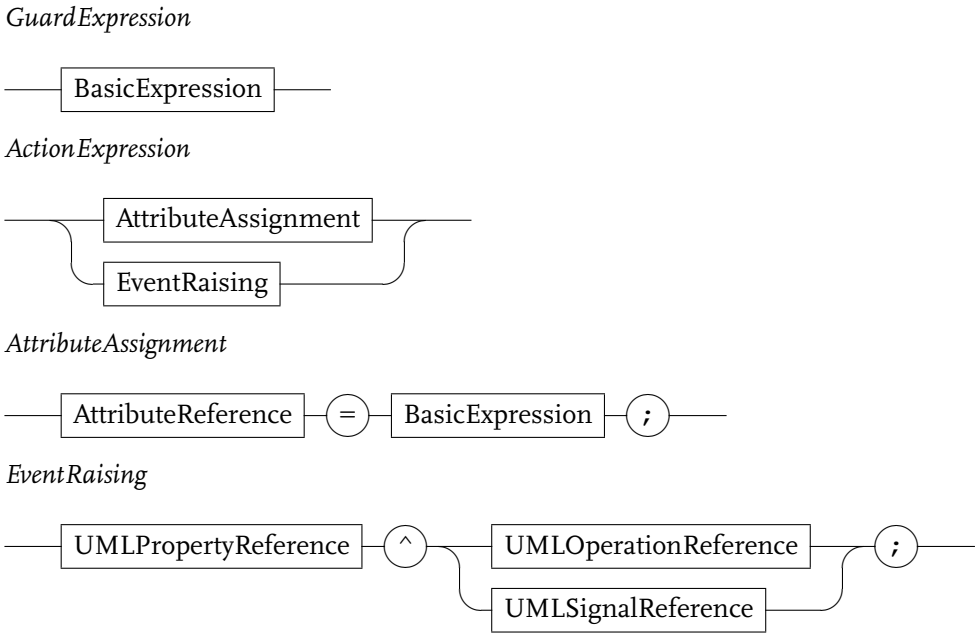


Figure A.2.: Rule for guard and action expressions used in UML statecharts

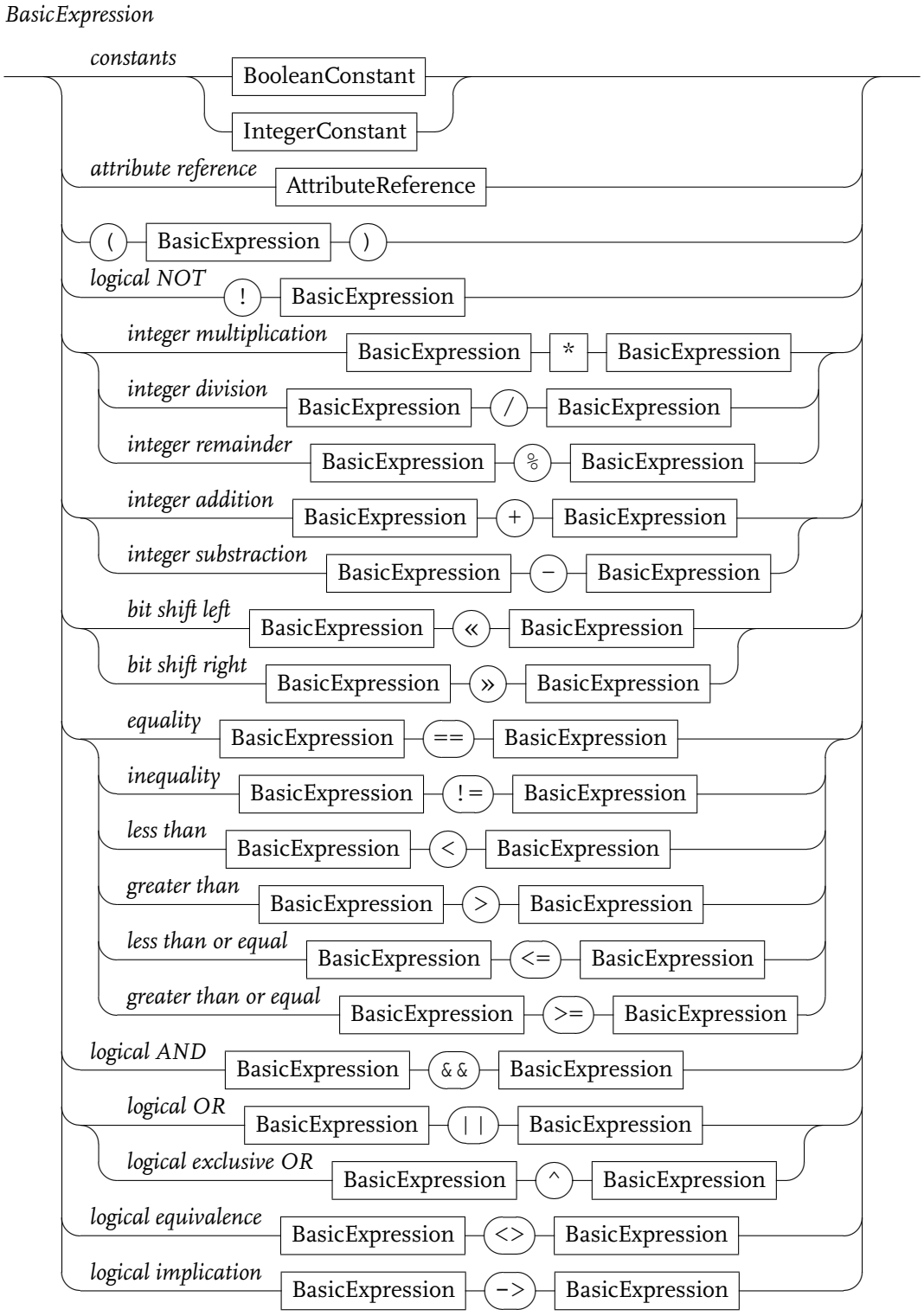


Figure A.3.: Rule for basic expressions used in UML statecharts. The order of the rules is equal to the order of the parsing precedence from high to low.

AttributeReference

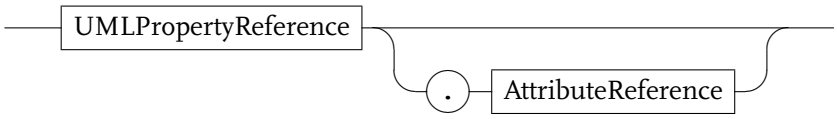
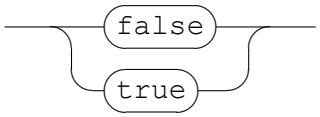


Figure A.4.: Rule for references to UML properties

BooleanConstant



IntegerConstant

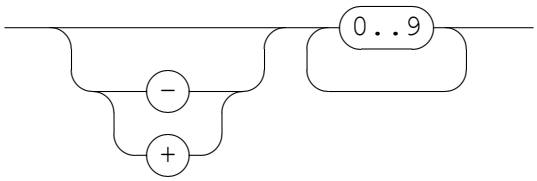


Figure A.5.: Rule for boolean and integer constants

APPENDIX B

User-Defined Xtend Functions

Listing B.1: User defined Model-2-model transformations for the ATN language.

```
1  temporalExpression::Parenthesis createParenthesis(mbtrs::Expression p_expression) :
2    JAVA org.haschemi.azmun.atn.ATNConstants.createParenthesis(org.haschemi.azmun.mbt.mbtrs.Expression);
3
4  temporalExpression::Binary createAnd(mbtrs::Expression p_left, mbtrs::Expression p_right) :
5    JAVA org.haschemi.azmun.atn.ATNConstants.createAnd(
6      org.haschemi.azmun.mbt.mbtrs.Expression, org.haschemi.azmun.mbt.mbtrs.Expression);
7
8  temporalExpression::Binary createOr(mbtrs::Expression p_left, mbtrs::Expression p_right) :
9    JAVA org.haschemi.azmun.atn.ATNConstants.createOr(
10     org.haschemi.azmun.mbt.mbtrs.Expression, org.haschemi.azmun.mbt.mbtrs.Expression);
11
12 temporalExpression::Binary createImplies(mbtrs::Expression p_left, mbtrs::Expression p_right) :
13   JAVA org.haschemi.azmun.atn.ATNConstants.createImplies(
14     org.haschemi.azmun.mbt.mbtrs.Expression, org.haschemi.azmun.mbt.mbtrs.Expression);
15
16 temporalExpression::Binary createEquiv(mbtrs::Expression p_left, mbtrs::Expression p_right) :
17   JAVA org.haschemi.azmun.atn.ATNConstants.createEquiv(
18     org.haschemi.azmun.mbt.mbtrs.Expression, org.haschemi.azmun.mbt.mbtrs.Expression);
19
20 temporalExpression::Unary createNot(mbtrs::Expression p_expression) :
21   JAVA org.haschemi.azmun.atn.ATNConstants.createNot(org.haschemi.azmun.mbt.mbtrs.Expression);
22
23 temporalExpression::Unary createGlobally(mbtrs::Expression p_expression) :
24   createUnary(p_expression, GLOBALLY());
25
```

```
26 temporalExpression::Unary createNext(mbtrs::Expression p_expression) : createUnary(p_expression, NEXT());
27
28 temporalExpression::Binary createBinary(mbtrs::Expression p_left, String p_feature, mbtrs::Expression p_right) :
29   JAVA org.haschemi.azmun.atn.ATNConstants.createBinary(
30     org.haschemi.azmun.mbt.mbtrs.Expression, java.lang.String, org.haschemi.azmun.mbt.mbtrs.Expression);
31
32 temporalExpression::Unary createUnary(mbtrs::Expression p_expression, String p_feature) :
33   JAVA org.haschemi.azmun.atn.ATNConstants.createUnary(
34     org.haschemi.azmun.mbt.mbtrs.Expression, java.lang.String);
35
36 temporalExpression::BooleanConstant createBooleanConstant(Boolean p_constant) :
37   JAVA org.haschemi.azmun.atn.ATNConstants.createBooleanConstant(java.lang.Boolean);
38
39 temporalExpression::IntegerConstant createIntegerConstant(Integer p_constant) :
40   JAVA org.haschemi.azmun.atn.ATNConstants.createIntegerConstant(java.lang.Integer);
41
42 temporalExpression::Variable createVariable(uml::NamedElement p_namedElement) :
43   JAVA org.haschemi.azmun.atn.ATNConstants.createVariable(org.eclipse.uml2.uml.NamedElement);
```

APPENDIX C

Reachability Tree Creation Algoritm

Listing C.1: Java-Code which builds the reachability tree using breadth-first search.

```
1  public final class ReachabilityGraphUtil {
2    public static ReachabilityGraph buildReachabilityGraph(final List<org.eclipse.uml2.uml.Class> p_comps) {
3      final List<Pseudostate> initalStates = collectInitialStatesFor(p_comps);
4
5      final ReachabilityGraph reachabilityGraph = MbtrsFactory.eINSTANCE.createReachabilityGraph();
6
7      final Queue<Configuration> agenda = new LinkedList<Configuration>();
8      // put initial configuration into the agenda
9      final Configuration initConfiguration
10       = MbtrsFactory.eINSTANCE.createConfiguration();
11
12     initConfiguration.getVertices().addAll(initalStates);
13     agenda.add(initConfiguration);
14     reachabilityGraph.getConfigurations().add(initConfiguration);
15     reachabilityGraph.setRoot(initConfiguration);
16
17     while (!agenda.isEmpty()) {
18       // get first element in agenda
19       final Configuration configuration = agenda.poll();
20
21       /* expand the current configuration by following all transitions in all vertices */
22       for (final Vertex vertex : configuration.getVertices()) {
23         for (final Transition transition : vertex.getOutgoings()) {
24           Configuration newConfig = getNextConfiguration(configuration, vertex, transition);
25
```

```
26            Configuration historyConfiguration = getHistoryConfigurationIfAny(newConfig, reachabilityGraph);
27
28            if (historyConfiguration == null) {
29              agenda.add(newConfig);
30              reachabilityGraph.getConfigurations().add(newConfig);
31            } else {
32              newConfig = historyConfiguration;
33            }
34
35            ConfigurationTransition configurationTransition = createConfigurationTransition(
36                reachabilityGraph, configuration, transition, newConfig);
37
38            reachabilityGraph.getConfigurationTransitions().add(configurationTransition);
39            configuration.getOutgoings().add(configurationTransition);
40          }
41        }
42      }
43      return reachabilityGraph;
44    }
45
46    private static ConfigurationTransition createConfigurationTransition(
47      final ReachabilityGraph p_reachabilityGraph,
48      final Configuration p_configuration,
49      final Transition p_transition,
50      Configuration p_newConfiguration) {
51
52      final ConfigurationTransition configurationTransition
53        = MbtrsFactory.eINSTANCE.createConfigurationTransition();
54
55      configurationTransition.setReferredTransition(p_transition);
56      configurationTransition.setSource(p_configuration);
57      configurationTransition.setTarget(p_newConfiguration);
58      return configurationTransition;
59    }
60
61    private static Configuration getHistoryConfigurationIfAny(Configuration p_newConfiguration,
62      ReachabilityGraph p_reachabilityGraph) {
63
64      for (final Configuration c : p_reachabilityGraph.getConfigurations()) {
65        if (ListUtils.isEqualList(c.getVertices(), p_newConfiguration.getVertices())) {
66          return c;
```

```
67          }
68        }
69        return null;
70      }
71
72      private static Configuration getNextConfiguration(final Configuration configuration, final Vertex vertex,
73        final Transition transition) {
74
75        final List<Vertex> newVertices = new LinkedList<Vertex>();
76        final Vertex newVertex = transition.getTarget();
77        newVertices.add(newVertex);
78
79        for (final Vertex v : configuration.getVertices()) {
80          boolean isSameContainer = v.getContainer() != null
81            && newVertex.getContainer() != null && v.getContainer().equals(newVertex.getContainer());
82
83          if (v.equals(vertex) || isSameContainer) {
84            continue;
85          }
86          newVertices.add(v);
87        }
88        if (isOrthogonalState(vertex)) {
89          for (final Region region : ((State) vertex).getRegions()) {
90            removeVerticesOfRegion(region, newVertices);
91          }
92        }
93
94        // if the current vertex is an AND-state, we add the initial vertices of each region to the configuration.
95        if (isOrthogonalState(newVertex)) {
96          for (final Region region : ((State) newVertex).getRegions()) {
97            // find initial vertex in region and add it to the
98            for (final Vertex v : region.getSubvertices()) {
99              if (v instanceof Pseudostate && ((Pseudostate) v).getKind() == PseudostateKind.INITIAL_LITERAL) {
100               newVertices.add(v);
101               break;
102             }
103           }
104         }
105       }
106
107       final Configuration newConfiguration = MbtrsFactory.eINSTANCE.createConfiguration();
```

```
108     newConfiguration.getVertices().addAll(newVertices);
109     return newConfiguration;
110   }
111
112   private static List<Pseudostate> collectInitialStatesFor(final List<Class> p_classes) {
113     // collect the initial states of the state machines of the classes
114     final List<Pseudostate> initialStates = new ArrayList<Pseudostate>(p_classes.size());
115
116     for (final Class clazz : p_classes) {
117       if (clazz.getOwnedBehaviors() == null) {
118         throw new RuntimeException("ownedBehaviour of class '" + clazz + "' is null.");
119       }
120
121       for (final Behavior behavior : clazz.getOwnedBehaviors()) {
122         if (behavior instanceof StateMachine) {
123           final StateMachine stateMachine = (StateMachine) behavior;
124           if (stateMachine.getRegions() == null) {
125             throw new RuntimeException("no regions in state machine '" + stateMachine + "'.");
126           }
127           boolean foundInitial = false;
128           for (final Vertex vertex : stateMachine.getRegions().get(0).getSubvertices()) {
129             if (vertex instanceof Pseudostate
130               && ((Pseudostate) vertex).getKind().equals(PseudostateKind.INITIAL_LITERAL)) {
131               foundInitial = true;
132               initialStates.add((Pseudostate) vertex);
133             }
134           }
135           if (!foundInitial) {
136             throw new RuntimeException("No initial state for state machine '" + stateMachine + "' found.");
137           }
138         }
139       }
140     }
141
142     return initialStates;
143   }
144
145   private static void removeVerticesOfRegion(final Region p_region, final List<Vertex> p_vertices) {
146     for (final Vertex v : p_region.getSubvertices()) {
147       p_vertices.remove(v);
148       if (v instanceof State) {
```

```
149             for (final Region r : ((State) v).getRegions()) {
150               removeVerticesOfRegion(r, p_vertices);
151             }
152           }
153         }
154       }
155
156       private static boolean isOrthogonalState(final Vertex p_vertex) {
157         return p_vertex instanceof State && !((State) p_vertex).getRegions().isEmpty();
158       }
159     }
```

APPENDIX D

Grammar for LTL Expressions used in Expression Test Case Specifications

LTLExpression

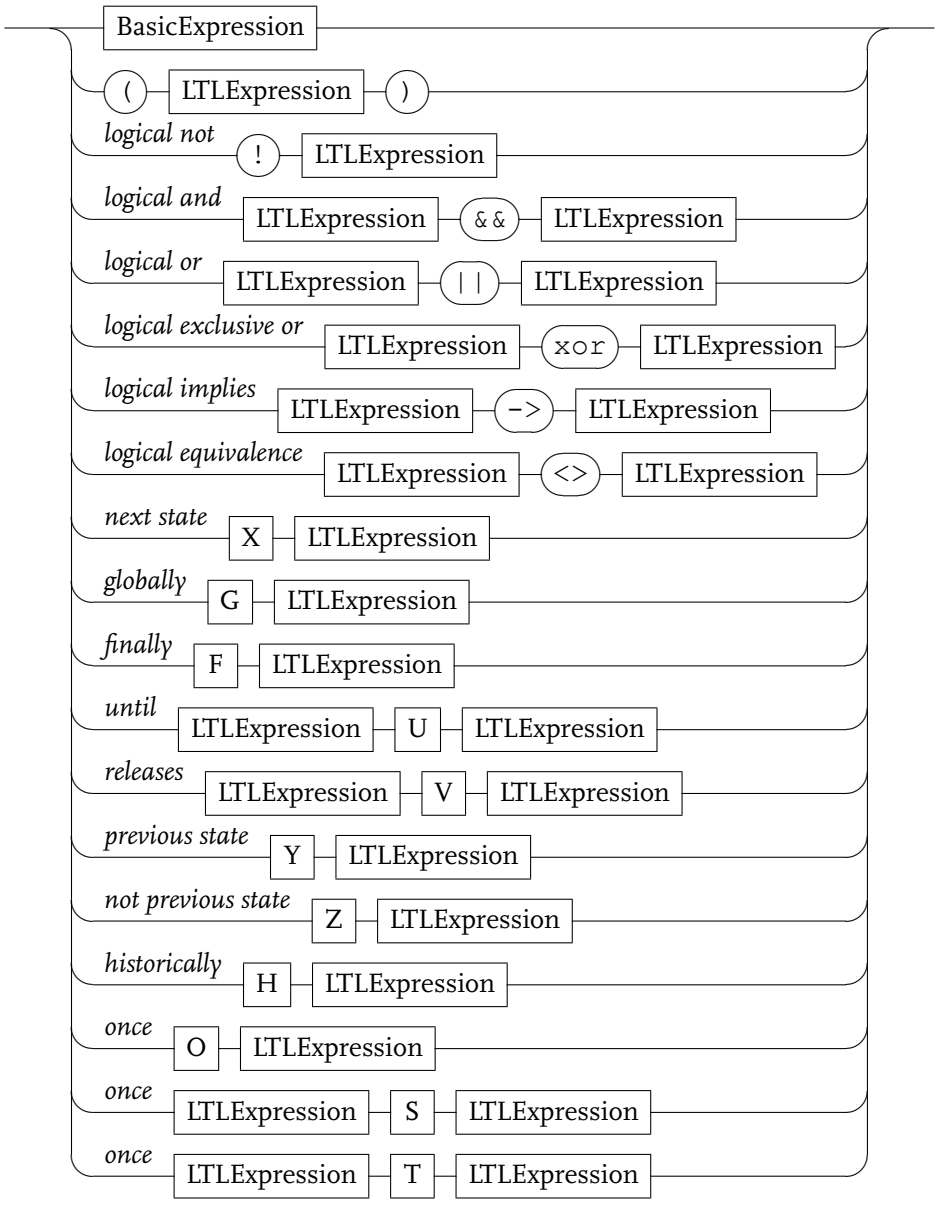


Figure D.1.: Rule for LTL expressions. The order of the rules is equal to the order of the parsing precedence from high to low.

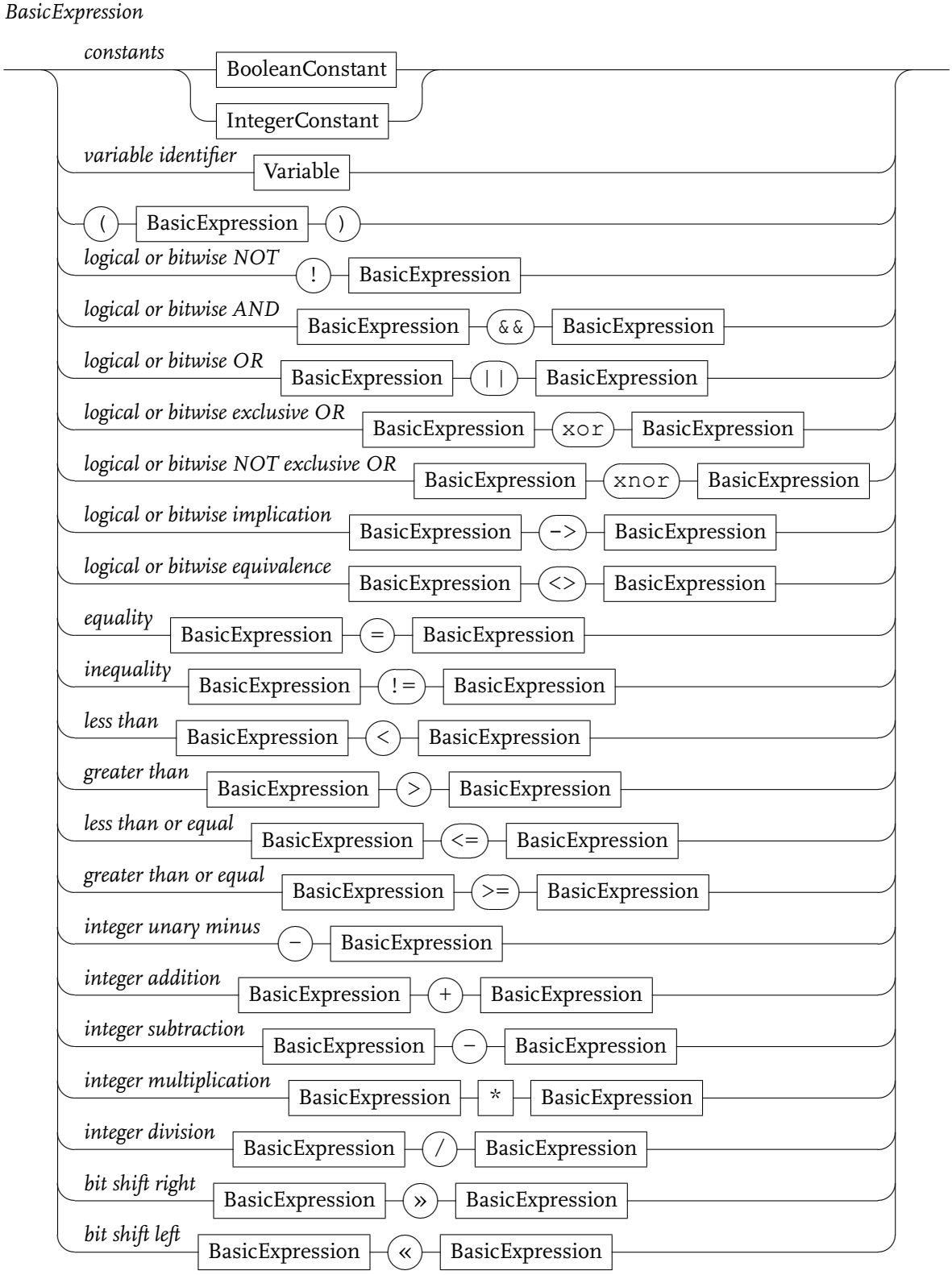


Figure D.2.: Rule for basic expressions used in LTL expressions. The order of the rules is equal to the order of the parsing precedence from high to low.

APPENDIX E

Generated NuSMV Model For The Online Storage Example

Listing E.1: Generated NuSMV model containing the main module. This module instantiates the User, Client, and Server modules and passes injects the dependencies into the modules.

```
1 MODULE main -- entry point
2
3 VAR
4
5 Server : process Server ( Client );
6 Client : process Client ( Server );
7 User : process User ( Client );
```

Listing E.2: Generated NuSMV model for the user component of the Online Storage Example 4.4.

```
1  MODULE User ( User#client )
2
3
4  VAR -- DECLARATIONS
5
6  User#Root_azmun_SUBSTATE : { nil, User#Root#init, User#Root#idle };
7  User#Root_azmun_TRANSITION : {
8    nil, User#Root#init_2_idle, User#Root#uiSendFile, User#Root#uiCancel
9  };
10
11 ASSIGN -- INITIALIZATION AND NEXT-STATE DEFINITIONS
```

```
12
13 -- Definition of Region Root
14 init(User#Root_azmun_SUBSTATE) := User#Root#init ;
15 init(User#Root_azmun_TRANSITION) := nil;
16
17 next ( User#client.queue[0] ) := case
18   (User#client.queue[0] = nil) : case -- CHECK IF THIS QUEUE-CELL IS THE FIRST FREE ONE
19     next(User#Root_azmun_TRANSITION)=User#Root#uiCancel : IClientUI#uiCancel;
20     next(User#Root_azmun_TRANSITION)=User#Root#uiSendFile : IClientUI#uiSendFile;
21    TRUE : nil;
22  esac;
23
24   TRUE : User#client.queue[0];
25 esac;
26
27 next(User#Root_azmun_TRANSITION) := case
28   (User#Root_azmun_SUBSTATE) = User#Root#init
29    : {
30       ( TRUE ? (User#Root#init_2_idle) : nil )
31    }; -- NON-DETERMINSIC CHOICE
32
33   (User#Root_azmun_SUBSTATE) = User#Root#idle
34     : {
35         ( TRUE & ( ! User#client . queueFull ) ? (User#Root#uiSendFile) : nil ),
36         ( TRUE & ( ! User#client . queueFull ) ? (User#Root#uiCancel) : nil )
37     }; -- NON-DETERMINSIC CHOICE
38
39   TRUE : nil;
40 esac;
41
42 next(User#Root_azmun_SUBSTATE) := case
43   next(User#Root_azmun_TRANSITION)=User#Root#init_2_idle: User#Root#idle;
44   next(User#Root_azmun_TRANSITION)=User#Root#uiSendFile: User#Root#idle;
45   next(User#Root_azmun_TRANSITION)=User#Root#uiCancel: User#Root#idle;
46   TRUE : User#Root_azmun_SUBSTATE;
47 esac;
```

Listing E.3: Generated NuSMV model for the client component of the Online Storage Example 4.4.

```
1 MODULE Client ( Client#storage )
```

```
2
3  VAR -- DECLARATIONS
4
5  queue: array 0..0 of { nil, IStorageClient#transferApproved, IStorageClient#transferCompleted,
6    IStorageClient#transferAborted, IClientUI#uiSendFile, IClientUI#uiCancel };
7
8  -- Definition of Region Root
9  Client#Root_azmun_SUBSTATE : { nil, Client#Root#init, Client#Root#idle,
10   Client#Root#sendFileReq, Client#Root#sendingFile, Client#Root#sendChoice };
11
12 Client#Root_azmun_TRANSITION : { nil, Client#Root#init_2_clientIdle, Client#Root#cancel_sendFileReq,
13   Client#Root#transferApproved, Client#Root#cancel_sendingFile, Client#Root#fileTransferCompleted,
14   Client#Root#fileCounterMaxReached, Client#Root#sendFileMaxNotReached, Client#Root#uiSendFile
15 };
16
17 Client#fileContingent : 0..10;
18
19
20 DEFINE -- MACROS
21
22 first   := queue[0] ;
23 queueFull  := queue[0] != nil;
24
25 currentTransConsumedEvent := Client#Root_azmun_TRANSITION=Client#Root#uiSendFile
26   | Client#Root_azmun_TRANSITION=Client#Root#cancel_sendingFile
27   | Client#Root_azmun_TRANSITION=Client#Root#cancel_sendFileReq
28   | Client#Root_azmun_TRANSITION=Client#Root#transferApproved
29   | Client#Root_azmun_TRANSITION=Client#Root#fileTransferCompleted
30 ;
31
32
33 ASSIGN -- INITIALIZATION AND NEXT-STATE DEFINITIONS
34
35 init(Client#Root_azmun_SUBSTATE) := Client#Root#init;
36 init(Client#Root_azmun_TRANSITION) := nil;
37 init(Client#fileContingent) := 10 ;
38
39 init (queue[ 0 ]) := nil;
40 next (queue[ 0 ]) := case
41  (next(currentTransConsumedEvent = TRUE)) : nil; -- SHIFT
42  TRUE : queue[0]; -- NOTHING CHANGED
```

```
43 esac;
44
45 next ( Client#storage.queue[0] ) := case
46   (Client#storage.queue[0] = nil) : case -- CHECK IF THIS QUEUE-CELL IS THE FIRST FREE ONE
47    next(Client#Root_azmun_TRANSITION)=Client#Root#sendFileMaxNotReached
48        : IStorageServer#requestSendFile;
49   next(Client#Root_azmun_TRANSITION)=Client#Root#cancel_sendingFile
50        : IStorageServer#cancelFile;
51   next(Client#Root_azmun_TRANSITION)=Client#Root#cancel_sendFileReq
52        : IStorageServer#cancelFile;
53   TRUE : nil;
54  esac;
55
56   TRUE : Client#storage.queue[0];
57 esac;
58
59
60 next(Client#Root_azmun_TRANSITION) := case
61   (Client#Root_azmun_SUBSTATE) = Client#Root#init
62     : {
63          ( TRUE ? (Client#Root#init_2_clientIdle) : nil )
64     }; -- NON-DETERMINSIC CHOICE
65
66   (Client#Root_azmun_SUBSTATE) = Client#Root#idle
67   & first = IClientUI#uiSendFile -- INVOLVED_SIGNAL
68     : ( TRUE ? (Client#Root#uiSendFile) : nil );
69
70   (Client#Root_azmun_SUBSTATE) = Client#Root#sendFileReq
71   & first = IClientUI#uiCancel -- INVOLVED_SIGNAL
72   : ( TRUE & ( !Client#storage.queueFull ) ? (Client#Root#cancel_sendFileReq) : nil );
73
74   (Client#Root_azmun_SUBSTATE) = Client#Root#sendFileReq
75   & first = IStorageClient#transferApproved -- INVOLVED_SIGNAL
76     : ( TRUE ? (Client#Root#transferApproved) : nil );
77
78   (Client#Root_azmun_SUBSTATE) = Client#Root#sendingFile
79   & first = IClientUI#uiCancel -- INVOLVED_SIGNAL
80     : ( TRUE & ( ! Client#storage . queueFull ) ? (Client#Root#cancel_sendingFile) : nil );
81
82   (Client#Root_azmun_SUBSTATE) = Client#Root#sendingFile
83   & first = IStorageClient#transferCompleted -- INVOLVED_SIGNAL
```

```
84      : ( TRUE & (Client#fileContingent − 1 in 0..10 ) ? (Client#Root#fileTransferCompleted) : nil );
85
86    (Client#Root_azmun_SUBSTATE) = Client#Root#sendChoice
87    & ( ! (Client#fileContingent > 0) )
88      : ( TRUE ? (Client#Root#fileCounterMaxReached) : nil );
89
90    (Client#Root_azmun_SUBSTATE) = Client#Root#sendChoice
91    & (Client#fileContingent > 0)
92      : ( TRUE & ( ! Client#storage . queueFull ) ? (Client#Root#sendFileMaxNotReached) : nil );
93
94    TRUE : nil;
95  esac;
96
97  next(Client#Root_azmun_SUBSTATE) := case
98    next(Client#Root_azmun_TRANSITION)=Client#Root#init_2_clientIdle
99      : Client#Root#idle;
100   next(Client#Root_azmun_TRANSITION)=Client#Root#cancel_sendFileReq
101     : Client#Root#idle;
102   next(Client#Root_azmun_TRANSITION)=Client#Root#cancel_sendingFile
103     : Client#Root#idle;
104   next(Client#Root_azmun_TRANSITION)=Client#Root#fileTransferCompleted
105     : Client#Root#idle;
106   next(Client#Root_azmun_TRANSITION)=Client#Root#fileCounterMaxReached
107     : Client#Root#idle;
108   next(Client#Root_azmun_TRANSITION)=Client#Root#sendFileMaxNotReached
109     : Client#Root#sendFileReq;
110   next(Client#Root_azmun_TRANSITION)=Client#Root#transferApproved
111     : Client#Root#sendingFile;
112   next(Client#Root_azmun_TRANSITION)=Client#Root#uiSendFile
113     : Client#Root#sendChoice;
114   TRUE : Client#Root_azmun_SUBSTATE;
115 esac;
116
117 next ( Client#fileContingent ) := case
118   next(Client#Root_azmun_TRANSITION)=Client#Root#fileTransferCompleted
119   & ((Client#fileContingent − 1) in (0..10))
120     : (Client#fileContingent − 1);
121
122   TRUE : Client#fileContingent;
123 esac;
```

Listing E.4: Generated NuSMV model for the server component of the Online Storage Example 4.4.

```
1  MODULE Server ( Server#client )
2
3  VAR -- DECLARATIONS
4  queue: array 0..0 of { nil, IStorageServer#requestSendFile, IStorageServer#cancelFile, IStorageServer#op1 };
5
6  -- Definition of Region Root
7  Server#Root_azmun_SUBSTATE : { nil, Server#Root#Server, Server#Root#init };
8  Server#Root_azmun_TRANSITION : { nil, Server#Root#init_2_server };
9
10 -- Definition of Region Main
11 Server#Root#Server#Main_azmun_SUBSTATE : { nil, Server#Root#Server#Main#init,
12   Server#Root#Server#Main#ReceiveState, Server#Root#Server#Main#idle };
13
14 Server#Root#Server#Main_azmun_TRANSITION : { nil, Server#Root#Server#Main#reqSendFile,
15   Server#Root#Server#Main#init_2_serverIdle, Server#Root#Server#Main#cancelFile,
16   Server#Root#Server#Main#diskFull };
17
18 -- Definition of Region Root
19 Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE : { nil,
20   Server#Root#Server#Main#ReceiveState#Root#receivingFile,
21   Server#Root#Server#Main#ReceiveState#Root#receivedFileReq,
22   Server#Root#Server#Main#ReceiveState#Root#init };
23
24 Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION : { nil,
25   Server#Root#Server#Main#ReceiveState#Root#init_2_receivedFileReq,
26   Server#Root#Server#Main#ReceiveState#Root#authenticated,
27   Server#Root#Server#Main#ReceiveState#Root#transferComplete };
28
29 -- Definition of Region DiskSpace
30 Server#Root#Server#DiskSpace_azmun_SUBSTATE : { nil, Server#Root#Server#DiskSpace#spaceAvailable,
31 Server#Root#Server#DiskSpace#diskFull, Server#Root#Server#DiskSpace#init };
32
33 Server#Root#Server#DiskSpace_azmun_TRANSITION : { nil,
34   Server#Root#Server#DiskSpace#init_2_SpaceAvailable,
35   Server#Root#Server#DiskSpace#discFull, Server#Root#Server#DiskSpace#discNotFull };
36
37 Server#transferCompleted : boolean;
38 Server#clientAuthenticated : boolean;
```

```
39 Server#discFull : boolean;
40
41
42 DEFINE -- MACROS
43 first  := queue[0] ;
44 queueFull  := queue[0] != nil;
45 currentTransConsumedEvent :=
46    Server#Root#Server#Main_azmun_TRANSITION=Server#Root#Server#Main#reqSendFile
47 | Server#Root#Server#Main_azmun_TRANSITION=Server#Root#Server#Main#cancelFile;
48
49
50 ASSIGN -- INITIALIZATION AND NEXT-STATE DEFINITIONS
51 init(Server#Root_azmun_SUBSTATE) := Server#Root#init;
52 init(Server#Root_azmun_TRANSITION) := nil;
53 init(Server#Root#Server#Main_azmun_SUBSTATE) := nil;
54 init(Server#Root#Server#Main_azmun_TRANSITION) := nil;
55 init(Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE) := nil;
56 init(Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION) := nil;
57 init(Server#Root#Server#DiskSpace_azmun_SUBSTATE) := nil;
58 init(Server#Root#Server#DiskSpace_azmun_TRANSITION) := nil;
59 init(Server#discFull) := FALSE;
60
61 init (queue[ o ]) := nil;
62 next (queue[ o ]) := case
63  (next(currentTransConsumedEvent = TRUE)) : nil; -- SHIFT
64  TRUE : queue[0]; -- NOTHING CHANGED
65 esac;
66
67 next ( Server#client.queue[0] ) := case
68    (Server#client.queue[0] = nil) : case -- CHECK IF THIS QUEUE-CELL IS THE FIRST FREE ONE
69       next(Server#Root#Server#Main_azmun_TRANSITION)
70          = Server#Root#Server#Main#diskFull : IStorageClient#transferAborted;
71     next(Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION)
72        = Server#Root#Server#Main#ReceiveState#Root#transferComplete : IStorageClient#transferCompleted;
73    next(Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION)
74       = Server#Root#Server#Main#ReceiveState#Root#authenticated : IStorageClient#transferApproved;
75    TRUE : nil;
76  esac;
77    TRUE : Server#client.queue[0];
78 esac;
79
```

```
80  next(Server#Root_azmun_TRANSITION) := case
81    (Server#Root_azmun_SUBSTATE) = Server#Root#init −− +_STATE
82    : {
83        ( TRUE ? (Server#Root#init_2_server) : nil )
84    }; −− NON−DETERMINSIC CHOICE
85    TRUE : nil;
86  esac;
87
88  next(Server#Root_azmun_SUBSTATE) := case
89    next(Server#Root_azmun_TRANSITION)=Server#Root#init_2_server
90      : Server#Root#Server;
91    TRUE : Server#Root_azmun_SUBSTATE;
92  esac;
93
94  next(Server#Root#Server#Main_azmun_TRANSITION) := case
95    (Server#Root_azmun_SUBSTATE) = Server#Root#Server
96    & (Server#Root#Server#Main_azmun_SUBSTATE) = Server#Root#Server#Main#init
97      : {
98          ( TRUE ? (Server#Root#Server#Main#init_2_serverIdle) : nil )
99        }; −− NON−DETERMINSIC CHOICE
100
101   (Server#Root_azmun_SUBSTATE) = Server#Root#Server
102   & (Server#Root#Server#Main_azmun_SUBSTATE) = Server#Root#Server#Main#ReceiveState
103   & first = IStorageServer#cancelFile −− INVOLVED_SIGNAL
104     : ( TRUE ? (Server#Root#Server#Main#cancelFile) : nil );
105
106   (Server#Root_azmun_SUBSTATE) = Server#Root#Server
107   & (Server#Root#Server#Main_azmun_SUBSTATE) = Server#Root#Server#Main#ReceiveState
108   & ( Server#discFull )
109     : ( TRUE & ( ! Server#client . queueFull ) ? (Server#Root#Server#Main#diskFull) : nil );
110
111   (Server#Root_azmun_SUBSTATE) = Server#Root#Server
112   & (Server#Root#Server#Main_azmun_SUBSTATE) = Server#Root#Server#Main#idle
113   & first = IStorageServer#requestSendFile −− INVOLVED_SIGNAL
114     : ( TRUE ? (Server#Root#Server#Main#reqSendFile) : nil );
115   TRUE : nil;
116 esac;
117
118
119 next(Server#Root#Server#Main_azmun_SUBSTATE) := case
120   next(Server#Root_azmun_SUBSTATE)=Server#Root#Server
```

```
121       & Server#Root#Server#Main_azmun_SUBSTATE=nil : Server#Root#Server#Main#init;
122
123     next(Server#Root_azmun_SUBSTATE)!=Server#Root#Server : nil;
124
125     next(Server#Root#Server#Main_azmun_TRANSITION)=Server#Root#Server#Main#reqSendFile
126       : Server#Root#Server#Main#ReceiveState;
127
128     next(Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION)
129       = Server#Root#Server#Main#ReceiveState#Root#transferComplete : Server#Root#Server#Main#idle;
130
131     next(Server#Root#Server#Main_azmun_TRANSITION)
132       = Server#Root#Server#Main#init_2_serverIdle : Server#Root#Server#Main#idle;
133
134     next(Server#Root#Server#Main_azmun_TRANSITION)
135       = Server#Root#Server#Main#cancelFile : Server#Root#Server#Main#idle;
136
137     next(Server#Root#Server#Main_azmun_TRANSITION)
138       = Server#Root#Server#Main#diskFull : Server#Root#Server#Main#idle;
139
140      TRUE : Server#Root#Server#Main_azmun_SUBSTATE;
141   esac;
142
143   next(Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION) := case
144     (Server#Root#Server#Main_azmun_SUBSTATE) = Server#Root#Server#Main#ReceiveState
145     & (Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE)
146       = Server#Root#Server#Main#ReceiveState#Root#receivingFile
147     & ( Server#transferCompleted )
148     :( TRUE & ( ! Server#client . queueFull )
149      ? (Server#Root#Server#Main#ReceiveState#Root#transferComplete) : nil );
150
151     (Server#Root#Server#Main_azmun_SUBSTATE) = Server#Root#Server#Main#ReceiveState
152     & (Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE)
153       = Server#Root#Server#Main#ReceiveState#Root#receivedFileReq
154     & (Server#clientAuthenticated) : ( TRUE &
155       ( ! Server#client . queueFull ) ?
156       (Server#Root#Server#Main#ReceiveState#Root#authenticated) : nil );
157
158     (Server#Root#Server#Main_azmun_SUBSTATE) = Server#Root#Server#Main#ReceiveState
159     & (Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE)
160      = Server#Root#Server#Main#ReceiveState#Root#init
161       : {
```

```
162         ( TRUE ? (Server#Root#Server#Main#ReceiveState#Root#init_2_receivedFileReq) : nil )
163       }; -- NON-DETERMINSIC CHOICE
164
165     TRUE : nil;
166   esac;
167
168
169   next(Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE) := case
170     next(Server#Root#Server#Main_azmun_SUBSTATE)=Server#Root#Server#Main#ReceiveState
171       & Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE=nil
172       : Server#Root#Server#Main#ReceiveState#Root#init;
173     next(Server#Root#Server#Main_azmun_SUBSTATE)!=Server#Root#Server#Main#ReceiveState
174       : nil;
175     next(Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION)
176      = Server#Root#Server#Main#ReceiveState#Root#authenticated
177       : Server#Root#Server#Main#ReceiveState#Root#receivingFile;
178
179     next(Server#Root#Server#Main#ReceiveState#Root_azmun_TRANSITION)
180      = Server#Root#Server#Main#ReceiveState#Root#init_2_receivedFileReq
181        : Server#Root#Server#Main#ReceiveState#Root#receivedFileReq;
182
183     TRUE : Server#Root#Server#Main#ReceiveState#Root_azmun_SUBSTATE;
184   esac;
185
186   next(Server#Root#Server#DiskSpace_azmun_TRANSITION) := case
187     (Server#Root_azmun_SUBSTATE) = Server#Root#Server
188     & (Server#Root#Server#DiskSpace_azmun_SUBSTATE) = Server#Root#Server#DiskSpace#spaceAvailable
189       : {
190           ( TRUE ? (Server#Root#Server#DiskSpace#discFull) : nil )
191         }; -- NON-DETERMINSIC CHOICE
192
193     (Server#Root_azmun_SUBSTATE) = Server#Root#Server
194     & (Server#Root#Server#DiskSpace_azmun_SUBSTATE) = Server#Root#Server#DiskSpace#diskFull
195     :{
196         ( TRUE ? (Server#Root#Server#DiskSpace#discNotFull) : nil )
197      }; -- NON-DETERMINSIC CHOICE
198
199     (Server#Root_azmun_SUBSTATE) = Server#Root#Server
200     & (Server#Root#Server#DiskSpace_azmun_SUBSTATE) = Server#Root#Server#DiskSpace#init
201       : {
202           ( TRUE ? (Server#Root#Server#DiskSpace#init_2_SpaceAvailable) : nil )
```

```
203       }; -- NON-DETERMINSIC CHOICE
204
205     TRUE : nil;
206   esac;
207
208   next(Server#Root#Server#DiskSpace_azmun_SUBSTATE) := case
209     next(Server#Root_azmun_SUBSTATE)=Server#Root#Server
210       & Server#Root#Server#DiskSpace_azmun_SUBSTATE=nil : Server#Root#Server#DiskSpace#init;
211    next(Server#Root_azmun_SUBSTATE)!=Server#Root#Server : nil;
212    next(Server#Root#Server#DiskSpace_azmun_TRANSITION)
213       = Server#Root#Server#DiskSpace#init_2_SpaceAvailable : Server#Root#Server#DiskSpace#spaceAvailable;
214    next(Server#Root#Server#DiskSpace_azmun_TRANSITION)
215       = Server#Root#Server#DiskSpace#discNotFull : Server#Root#Server#DiskSpace#spaceAvailable;
216    next(Server#Root#Server#DiskSpace_azmun_TRANSITION)
217       = Server#Root#Server#DiskSpace#discFull : Server#Root#Server#DiskSpace#diskFull;
218    TRUE : Server#Root#Server#DiskSpace_azmun_SUBSTATE;
219   esac;
220
221   next ( Server#discFull ) := case
222     next(Server#Root#Server#DiskSpace_azmun_TRANSITION)
223       = Server#Root#Server#DiskSpace#discNotFull : (FALSE);
224     next(Server#Root#Server#DiskSpace_azmun_TRANSITION)
225       = Server#Root#Server#DiskSpace#discFull : (TRUE);
226     TRUE : Server#discFull;
227   esac;
```

APPENDIX F

Workflow Definition File

Listing F.1: XML based workflow definition of the common workflow for automatic test generation with model checkers(Fig. 4.19).

```
<?xml version="1.0"?>
<workflow>
  <property name="BASE_DIR" value="."/>
  <property name="MODEL_FILE" value=""/>
  <property name="MODEL_SLOT" value=""/>
  <property name="PROFILE_FILE" value="DCSP.profile.uml"/>
  <property name="GENERATION_DIRECTORY" value="${BASE_DIR}/src-gen"/>
  <property name="CLEAN_GENERATION_DIRECTORY" value="false"/>
  <property name="MODEL_CHECKER_MODEL_FILE" value="model.nusmv"/>
  <property name="BUFFER_SIZE" value="0"/>
  <property name="USE_FAIRNESS" value="false"/>
  <property name="FORMAT_OUTPUT" value="false"/>
  <property name="EXPORT_MAGIC_DRAW_MODEL" value="false"/>
  <property name="MAGIC_DRAW_PROJECT_FILE" value=""/>
  <property name="MAGIC_DRAW_EMFXMI_EXE" value=""/>
  <property name="MAGIC_DRAW_EXPORT_DESTINATION_DIR" value=""/>
  <property name="PREPARE_UML_MODEL" value="true"/>
  <property name="HUMAN_READABLE_NAMES" value="true"/>
  <property name="CHECK_MODEL" value="true"/>
  <property name="REMOVE_GENERATED_TEST_CASES" value="true"/>
  <property name="TESTGENERATION_ADAPTIVE" value="true"/>
  <property name="ADD_SELF_TRANSITIONS" value="false"/>
  <property name="USE_CONE_OF_INFLUENCE" value="false"/>
```

```
24    <property name="AG_ONLY_SEARCH" value="false"/>
25    <property name="MODEL_CHECKER_MODEL_GENERATION" value="true"/>
26    <property name="COMPUTE_TEST_CASE_SPECIFICATIONS" value="true"/>
27    <property name="PRORIZE_TEST_CASE_SPECIFICATIONS" value="false"/>
28    <property name="USE_EXPRESSIONS" value="false"/>
29    <property name="ABSTRACT_TEST_SUITE_GENERATION" value="true"/>
30    <property name="OPTIMIZE_TEST_SUITE" value="false"/>
31    <property name="CONCRETE_TEST_SUITE_GENERATION" value="true"/>
32    <property name="PRINT_STATISTICS" value="true"/>
33
34    <!-- The OSGi filters to control service selection -->
35    <property name="EXPORT_MAGIC_DRAW_MODEL_FILTER" value="(category=core)"/>
36    <property name="MODEL_CHECKER_MODEL_GENERATION_FILTER" value="(category=core)"/>
37    <property name="COMPUTE_TEST_CASE_SPECIFICATIONS_FILTER" value="(category=core)"/>
38    <property name="PRORIZE_TEST_CASE_SPECIFICATIONS_FILTER" value="(category=core)"/>
39    <property name="ABSTRACT_TEST_SUITE_GENERATION_FILTER" value="(category=core)"/>
40    <property name="OPTIMIZE_TEST_SUITE_FILTER" value="(category=core)"/>
41    <property name="CONCRETE_TEST_SUITE_GENERATION_FILTER" value="(category=core)"/>
42    <property name="PRINT_STATISTICS_FILTER" value="(category=core)"/>
43    <property name="MODEL_WRITER_FILTER" value="(category=core)"/>
44    <property name="CHECK_MODEL_FILTER" value="(category=core)"/>
45
46    <if cond="${CHECK_MODEL}">
47      <component id="Check Model" class="org.haschemi.azmun.workflow.ICheckModel"
48        serviceDiscovery="true" filter="${CHECK_MODEL_FILTER}">
49        <modelSlot value="${MODEL_SLOT}"/>
50        <checkFile value="org/haschemi/azmun/workflow/impl/checkAzmunModel"/>
51        <checkFile value="org/haschemi/azmun/workflow/impl/checkUMLModel"/>
52      </component>
53    </if>
54
55    <if cond="${PREPARE_UML_MODEL}">
56      <cartridge
57  file="org/haschemi/azmun/workflow/impl/prepareUMLModel.mwe"
58  inheritAll="true"/>
59    </if>
60
61    <if cond="${ADD_SELF_TRANSITIONS}">
62      <cartridge
63      file="org/haschemi/azmun/workflow/impl/addSelfTransitions.mwe"
64      inheritAll="true"/>
```

```
65  </if>
66
67  <if cond="${HUMAN_READABLE_NAMES}">
68    <cartridge
69    file="org/haschemi/azmun/workflow/impl/addHumanReadableNames.mwe"
70    inheritAll="true"/>
71  </if>
72
73  <if cond="${CLEAN_GENERATION_DIRECTORY}">
74    <component id="clean.output.directory" class="org.eclipse.emf.mwe.utils.DirectoryCleaner">
75      <directory value="${GENERATION_DIRECTORY}"/>
76    </component>
77  </if>
78
79  <if cond="${COMPUTE_TEST_CASE_SPECIFICATIONS}">
80    <component id="Compute Test Case Specifications"
81      class="org.haschemi.azmun.workflow.ITestCaseSpecificationComputation"
82      serviceDiscovery="true" filter="${COMPUTE_TEST_CASE_SPECIFICATIONS_FILTER}">
83      <modelSlot value="${MODEL_SLOT}"/>
84      <profile value="${PROFILE_FILE}"/>
85      <useExpressions value="${USE_EXPRESSIONS}"/>
86    </component>
87  </if>
88
89  <if cond="${PRORIZE_TEST_CASE_SPECIFICATIONS}">
90    <component id="Priorize Test Case Specifications"
91       class="org.haschemi.azmun.workflow.ITestCaseSpecificationPriorization"
92       serviceDiscovery="true" cardinality="0..1" filter="${PRORIZE_TEST_CASE_SPECIFICATIONS_FILTER}">
93      <modelSlot value="${MODEL_SLOT}"/>
94    </component>
95  </if>
96
97  <if cond="${MODEL_CHECKER_MODEL_GENERATION}">
98    <component id="Pre Test Generation"
99      class="org.haschemi.azmun.workflow.IModelCheckerModelGeneration"
100     serviceDiscovery="true" filter="${MODEL_CHECKER_MODEL_GENERATION_FILTER}">
101     <modelSlot value="${MODEL_SLOT}"/>
102     <fileUri value="${MODEL_CHECKER_MODEL_FILE}" />
103     <outputPath value="${GENERATION_DIRECTORY}"/>
104     <bufferSize value="${BUFFER_SIZE}" />
105     <useFairness value="${USE_FAIRNESS}"/>
```

```
106         <formatOutput value="${FORMAT_OUTPUT}"/>
107       </component>
108     </if>
109
110     <if cond="${ABSTRACT_TEST_SUITE_GENERATION}">
111       <component id="Abstract Test Suite Generation"
112         class="org.haschemi.azmun.workflow.IAbstractTestSuiteGeneration"
113         serviceDiscovery="true" filter="${ABSTRACT_TEST_SUITE_GENERATION_FILTER}">
114         <modelSlot value="${MODEL_SLOT}"/>
115         <fileUri value="${MODEL_CHECKER_MODEL_FILE}" />
116         <outputPath value="${GENERATION_DIRECTORY}"/>
117         <adaptive value="${TESTGENERATION_ADAPTIVE}"/>
118         <useConeOfInfluence value="${USE_CONE_OF_INFLUENCE}"/>
119         <agOnlySearch value="${AG_ONLY_SEARCH}"/>
120         <removeGeneratedTestCases value="${REMOVE_GENERATED_TEST_CASES}"/>
121       </component>
122     </if>
123
124     <if cond="${OPTIMIZE_TEST_SUITE}">
125       <component id="Abstract Test Suite Optimization"
126         class="org.haschemi.azmun.workflow.ITestSuiteOptimization"
127         serviceDiscovery="true" filter="${OPTIMIZE_TEST_SUITE_FILTER}">
128         <modelSlot value="${MODEL_SLOT}"/>
129       </component>
130     </if>
131
132     <component id="Azmun Model Writer"
133       class="org.haschemi.azmun.workflow.IModelWriter"
134       serviceDiscovery="true" cardinality="1..1" filter="${MODEL_WRITER_FILTER}">
135       <modelSlot value="${MODEL_SLOT}"/>
136     </component>
137 </workflow>
```

Bibliography

Acharya, A.A., Mohapatra, D.P., Panda, N., 2010. *Model based test case prioritization for testing component dependency in cbsd using uml sequence diagram*. International Journal of Advanced Computer Science and Applications-IJACSA 1(6).

Ahrens, K., Eveslage, I., Fischer, J., Kühnlenz, F., Weber, D., 2009. *The Challenges of Using SDL for the Development of Wireless Sensor Networks*. In R. Reed, A. Bilgic, R. Gotzhein (editors) *SDL 2009: Design for Motes and Mobiles*, volume 5719 of *Lecture Notes in Computer Science*, 200–221. Springer Berlin Heidelberg. ISBN 978-3-642-04553-0.

Akyildiz, I.F., Wang, X., Wang, W., 2005. *Wireless mesh networks: a survey*. Computer Networks 47(4):445–487.

Alda, S., Won, M., Cremers, A.B., 2003. *Managing Dependencies in Component-Based Distributed Applications*. In *In Revised Papers from the International Workshop on Scientific Engineering for Distributed Java Applications*, 143–154. Springer-Verlag.

Ammann, P., Offutt, J., 2008. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA. ISBN 9780521880381.

Ammann, P., Ding, W., Xu, D., 2001. *Using a Model Checker to Test Safety Properties*. In *In: Proceedings of the 7th International Conference on Engineering of Complex Computer Systems*, 212–221. IEEE.

Ammann, P.E., Black, P.E., Majurski, W., 1998. *Using model checking to generate tests from specifications*. In *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98*, 46–54. IEEE Computer Society.

AndroMDA, 2013. [Online]. Available: http://andromda.org/. [Accessed: Dez. 17, 2013].

Antoniol, G., Briand, L., Penta, M.D., Labiche, Y., 2002. *A Case Study Using the Round-Trip Strategy for State-Based Class Testing*. Software Reliability Engineering, International Symposium on 0:269. ISSN 1071-9458.

Arcaini, P., Gargantini, A., Riccobene, E., 2011. *A model advisor for NuSMV specifications*. Innovations in Systems and Software Engineering 7(2):97–107. ISSN 1614-5046.

Arcaini, P., Gargantini, A., Riccobene, E., 2013a. *Combining Model-Based Testing and Runtime Monitoring for Program Testing in the Presence of Nondeterminism*. In *ICST Workshops*, 178–187.

Arcaini, P., Gargantini, A., Vavassori, P., 2013b. *NuSeen: an eclipse-based environment for the NuSMV model checker*. In *Eclipse-IT 2013: Proceedings of VIII Workshop of the Italian Eclipse Community*.

Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., Washington, R., ???? *Automated Testing using Symbolic Model Checking and Temporal Monitoring*.

Basanieri, F., Bertolino, A., 2000. *A Practical Approach to UML-based Derivation of Integration Tests*. In *4th International Software Quality Week Europe*.

Beizer, B., 1990. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA. ISBN 0-442-20672-0.

Binder, R.V., 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc. ISBN 0201809389.

Boronat, A., Meseguer, J., 2008. *An algebraic semantics for MOF*. In *Fundamental Approaches to Software Engineering*, 377–391. Springer.

Bowen, J., 1998. *Select Z Bibliography*. In J. Bowen, A. Fett, M. Hinchey (editors) *ZUM '98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, 15–43. Springer Berlin / Heidelberg. ISBN 978-3-540-65070-6.

Briones, L.B., Brinksma, E., Stoelinga, M., 2006. *A Semantic Framework for Test Coverage*. In *Proceedings of the 4th International Conference on Automated Technology for Verification and Analysis*, ATVA'06, 399–414. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-47237-1, 978-3-540-47237-7.

Brown, A., 2000. *Large-scale, component-based development*. Object and component technology series. Prentice Hall PTR. ISBN 9780130887207.

Broy, M., Jonsson, B., Katoen, J.P., 2005. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer. ISBN 3540262784.

Callahan, J., Schneider, F., Easterbrook, S., 1996. *Automated software testing using model-checking*. In *Proceedings 1996 SPIN workshop*, volume 353. Citeseer.

Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T., 2006. *Managing the evolution of dataflows with vistrails*. In *IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow)*.

Calvagna, A., Gargantini, A., 2010. *A Formal Logic Approach to Constrained Combinatorial Testing*. Journal of Automated Reasoning ISSN 0168-7433.

Cervantes, H., Hall, R.S., 2003. *Automating Service Dependency Management in a Service-Oriented Component Model*. In *6th Workshop on Component-Based Software Engineering*, 379–382.

Cichos, H., Oster, S., Lochau, M., Schürr, A., 2011. *Model-based Coverage-Driven Test Suite Generation for Software Product Lines*. In *Proceedings of the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 6981 of *Lecture Notes in Computer Science (LNCS)*, 425–439.

Cichos, H., Lochau, M., Oster, S., Schürr, A., 2012. *Reduktion von Testsuiten für Software-Produktlinien*. In *Proceedings of the Software Engineering 2012*, GI-Edition Lecture Notes in Informatics, 143–154.

Cichos, H., 2013. *Modellbasierte Generierung und Reduktion von Testsuiten für Software-Produktlinien*. Ph.D. thesis, Technische Unterstität Darmstadt.

Cichos, H., Heinze, T.S., 2011. *Efficient Test Suite Reduction by Merging Pairs of Suitable Test Cases*. In *Proceedings of the 2010 International Conference on Models in Software Engineering*, MODELS'10, 244–258. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-21209-3.

Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M., 1999. *NuSMV: A new symbolic model verifier*. 495–499. Springer.

Clarke, E.M., Emerson, E.A., Sistla, A.P., 1986. *Automatic verification of finite-state concurrent systems using temporal logic specifications.* ACM Trans. Program. Lang. Syst. 8(2):244–263. ISSN 0164-0925.

Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J., 1985. *A Comparison of Data Flow Path Selection Criteria.* In *Proceedings of the 8th International Conference on Software Engineering*, ICSE '85, 244–251. IEEE Computer Society Press, Los Alamitos, CA, USA. ISBN 0-8186-0620-7.

Czarnecki, K., Helsen, S., 2006. *Feature-based survey of model transformation approaches.* IBM Syst. J. 45:621–645. ISSN 0018-8670.

Czarnecki, K., Helsen, S., 2003. *Classification of Model Transformation Approaches.*

Dashofy, E.M., van der Hoek, A., Taylor, R.N., 2002. *Towards Architecture-based Self-healing Systems.* In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, 21–26. ACM, New York, NY, USA. ISBN 1-58113-609-9.

Deelman, E., Gannon, D., Shields, M., Taylor, I., 2009. *Workflows and e-Science: An overview of workflow system features and capabilities.* Future Generation Computer Systems 25(5):528 – 540. ISSN 0167-739X.

Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H., 2007. *A survey on model-based testing approaches: a systematic review.* In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, 31–36. ACM, New York, NY, USA. ISBN 978-1-59593-880-0.

Drusinsky, D., 2006. *Modeling and Verification Using UML Statecharts, First Edition : A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking.* Newnes. ISBN 0750679492.

Eclipse, 2012. *Xpand / Xtend Reference.*

Eisenbach, S., Sadler, C., Shaikh, S., 2002. *Evolution of Distributed Java Programs.* In *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, CD '02, 51–66. Springer-Verlag, London, UK, UK. ISBN 3-540-43847-5.

Engels, A., Feijs, L., Mauw, S., 1997. *Test generation for intelligent networks using model checking*. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97), volume 1217 of Lecture Notes in Computer Science*, 384–398. Springer.

Faivre, A., Gaston, C., Gall, P., 2007. *Symbolic Model Based Testing for Component Oriented Systems*. In A. Petrenko, M. Veanes, J. Tretmans, W. Grieskamp (editors) *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, 90–106. Springer Berlin Heidelberg. ISBN 978-3-540-73065-1.

Fischer, J., Kühnlenz, F., Ahrens, K., Eveslage, I., 2009. *Model-based Development of Self-organizing Earthquake Early Warning Systems*. In I. Troch, F. Breitenecker (editors) *Proceedings MathMod Vienna 2009*, number 35 in Argesim Report. Vienna University of Technology.

Fischer, J., Redlich, J.P., Zschau, J., Milkereit, C., Picozzi, M., Fleming, K., Brumbulli, M., Lichtblau, B., Eveslage, I., 2012. *A wireless mesh sensing network for early warning*. Journal of Network and Computer Applications 35(2):538 – 547. ISSN 1084-8045.

Fraser, G., 2007. *Automated Software Testing with Model Checkers*. Ph.D. thesis, ST - Institute for Softwaretechnology Graz University of Technology.

Fraser, G., Wotawa, F., 2007a. *Nondeterministic Testing with Linear Model-Checker Counterexamples*. In *Proceedings of the Seventh International Conference on Quality Software*, QSIC '07, 107–116. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-3035-4.

Fraser, G., Wotawa, F., 2007b. *Using LTL rewriting to improve the performance of model-checker based test-case generation*. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, A-MOST '07, 64–74. ACM, New York, NY, USA. ISBN 978-1-59593-850-3.

Fraser, G., Wotawa, F., 2008a. *Ordering Coverage Goals in Model Checker Based Testing*. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, 31–40. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3388-9.

Fraser, G., Wotawa, F., 2008b. *Using Model-checkers to Generate and Analyze Property Relevant Test-cases*. Software Quality Control 16(2):161–183. ISSN 0963-9314.

Fraser, G., Wotawa, F., Ammann, P.E., 2009. *Testing with model checkers: a survey*. Softw. Test. Verif. Reliab. 19(3):215–261. ISSN 0960-0833.

Friedenthal, S., Moore, A., Steiner, R., 2008. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0123743796, 9780123743794.

Gargantini, A., Heitmeyer, C., 1999. *Using Model Checking to Generate Tests from Requirements Specifications*. In O. Nierstrasz, M. Lemoine (editors) *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, 146–162. Springer Berlin Heidelberg. ISBN 978-3-540-66538-0.

Gill, A., 1962. *Introduction to the Theory of Finite State Machines*. McGraw-Hill.

Gonzalez-Sanchez, A., Piel, E., Gross, H.G., van Gemund, A.J., 2010. *Runtime testability in dynamic high-availability component-based systems*. In *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on*, 37–42. IEEE.

Gross, H.G., 2004. *Component-based Software Testing With Uml*. SpringerVerlag. ISBN 354020864X.

Harel, D., Rumpe, B., 2004. *Meaningful modeling: what's the semantics of "semantics"?* Computer 37(10):64–72. ISSN 0018-9162.

Harel, D., 1987. *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program. 8(3):231–274. ISSN 0167-6423.

Harel, D., Politi, M., 1998. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA. ISBN 0070262055.

Haschemi, S., Wider, A., 2009. *An Extensible Language for Service Dependency Management*. In *35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*.

Heimdahl, M.P.E., George, D., 2004. *Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing*. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, 176–185. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2131-2.

Heinecke, H., Schnelle, K., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Maté, J., Nishikawa, K., Scharnhorst, T., 2004. *Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e architectures* .

Heineman, G.T., Councill, W.T. (editors) , 2001. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-70485-4.

Hollingsworth, D., et al., 1994. *Workflow management coalition: The workflow reference model*. Document TC00-1003, Workflow Management Coalition, Dec .

Holzmann, G.J., 1997. *The Model Checker SPIN*. IEEE Trans. Softw. Eng. 23(5):279–295. ISSN 0098-5589.

Hong, H.S., Ural, H., 2005. *Using Model Checking for Reducing the Cost of Test Generation*. In J. Grabowski, B. Nielsen (editors) *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, 110–124. Springer Berlin / Heidelberg.

Hong, H.S., Lee, I., Sokolsky, O., Cha, S.D., 2001. *Automatic Test Generation from Statecharts Using Model Checking*. In *In Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series*, 15–30.

Howes, T., 1996. *A String Representation of LDAP Search Filters*. RFC 1960 (Proposed Standard).

ISO/IEC, 1995. *ISO/IEC ISO 8402:1995-08. Quality management and quality assurance - Vocabulary*. ISO/IEC.

ISO/IEC, 2001. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC.

ISO/IEC, 2012. *ISO/IEC 25010:2011 . Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. ISO/IEC.

Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading.

Java Emitter Templates (JET), 2013. [Online]. See JET Tutorial by Remko Pompa at $www.eclipse.org/articles/Article-JET2/jet_tutorial2.html$. [Accessed: Dez. 17, 2013].

Kadono, M., Tsuchiya, T., Kikuno, T., 2009. *Using the NuSMV Model Checker for Test Generation from Statecharts.* Pacific Rim International Symposium on Dependable Computing, IEEE 0:37–42.

Kertész, A., Sipos, G., Kacsuk, P., 2007. *Brokering multi-grid workflows in the P-GRADE portal.* In *Proceedings of the CoreGRID 2006, UNICORE Summit 2006, Petascale Computational Biology and Bioinformatics conference on Parallel processing*, Euro-Par'06, 138–149. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-72226-7.

Klus, H., Niebuhr, D., Rausch, A., 2007. *A Component Model for Dynamic Adaptive Systems.* In *International Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ESSPE '07, 21–28. ACM, New York, NY, USA. ISBN 978-1-59593-798-8.

Kolach, O., 2011. *Modellbasiertes Testen: Implementierung und Anwendung von Mutationsanalyse-Techniken. Diploma Thesis.*

Krakowiak, S., 2007. *Middleware Architecture with Patterns and Frameworks.*

Kramer, J., Magee, J., 1990. *The Evolving Philosophers Problem: Dynamic Change Management.* IEEE Trans. Softw. Eng. 16(11):1293–1306. ISSN 0098-5589.

Kuliamin, V.V., Petrenko, A.K., Kossatchev, A.S., Bourdonov, I.B., 2003. *UniTesK: Model based testing in industrial practice.* In *1st European Conference on Model Driven Software Engineering*, 55–63.

Lam, V.S.W., 2006. *A Formal Execution Semantics and Rigorous Analytical Approach for Communicating Uml Statechart Diagrams.* Ph.D. thesis.

Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A., 2005. *Testing real-time embedded software using UPPAAL-TRON: an industrial case study.* In W. Wolf (editor) *EMSOFT*, 299–306. ACM. ISBN 1-59593-091-4.

Lasalle, J., Peureux, F., Fondement, F., 2011. *Development of an automated MBT toolchain from UML/SysML models.* Innovations in Systems and Software Engineering 7(4):247–256. ISSN 1614-5046.

Marré, M., Bertolino, A., 1996. *Reducing and estimating the cost of test coverage criteria.* In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, 486–494. IEEE Computer Society, Washington, DC, USA. ISBN 0-8186-7246-3.

Masiero, P.C., Maldonado, J.C., Bonaventura, I.G., 1994. *A reachability tree for statecharts and analysis of some properties.* Information and Software Technology 36(10):615–624.

Mayerhofer, T., Langer, P., Wimmer, M., 2012. *Towards xMOF: Executable DSMLs Based on fUML.* In *Proceedings of the 2012 Workshop on Domain-specific Modeling*, DSM '12, 1–6. ACM, New York, NY, USA. ISBN 978-1-4503-1634-7.

McIlroy, M.D., 1968. *Mass Produced Software Components.* In *Software Engineering, Report on a Conference sponsored by the NATO Science Committee. Garmisch, Germany.*

Microsoft, 2013. *Component Object Model, http://www.microsoft.com/com/default.mspx,.*

Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A., 2004. *SAL 2.* In R. Alur, D. Peled (editors) *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, 496–500. Springer Berlin Heidelberg. ISBN 978-3-540-22342-9.

Nayak, A., Samanta, D., 2009. *Model-based test cases synthesis using UML interaction diagrams.* SIGSOFT Softw. Eng. Notes 34:1–10. ISSN 0163-5948.

Object Management Group, 2003. *Meta Object Facility (MOF) 2.0 Core Specification, OMG document ptc/04-10-15.* OMG.

Object Management Group, 2005a. *UML Testing Profile , Version 1.0, OMG document formal/05-07-07.* OMG.

Object Management Group, 2005b. *Unified Modeling Language: Superstructure, version 2.0.* OMG.

Object Management Group, 2006a. *CORBA Component Model 4.0 Specification.* Specification Version 4.0, OMG.

Object Management Group, 2006b. *Object Constraint Language (OCL) 2.0, OMG document formal/2006-05-01.* OMG.

Object Management Group, 2009. *Model-Driven Architecture.* http://www.omg.org/mda/.

Oinn, T., Addis, M., Ferris, J., Marvin, D., Carver, T., Pocock, M.R., Wipat, A., 2004. *Taverna: A tool for the composition and enactment of bioinformatics workflows.* Bioinformatics 20:2004.

OMG, 2011. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1.*

Oracle, 2013. *Enterprise Java Beans Specification, JSR 345, Version 3.2.*

Oreizy, P., Medvidovic, N., Taylor, R.N., 1998. *Architecture-based Runtime Software Evolution.* In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, 177–186. IEEE Computer Society, Washington, DC, USA. ISBN 0-8186-8368-6.

OSGi Alliance, 2007. *OSGi Service Platform Core Specification, V4.1.* OSGi Alliance.

Paige, R.F., Kolovos, D.S., Polack, F.A., 2006. *An action semantics for MOF 2.0.* In *Proceedings of the 2006 ACM symposium on Applied computing*, 1304–1305. ACM.

Peleska, J., Vorobev, E., Lapschies, F., 2011. *Automated Test Case Generation with SMT-Solving and Abstract Interpretation.* In *NASA Formal Methods*, 298–312.

Poole, J., Mellor, D., 2001. *Common Warehouse Metamodel: An Introduction to the Standard for Data Warehouse Integration.* John Wiley & Sons, Inc., New York, NY, USA. ISBN 0471200522.

Queille, J., Sifakis, J., 1982. *Specification and verification of concurrent systems in CESAR.* In M. Dezani-Ciancaglini, U. Montanari (editors) *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, 337–351. Springer Berlin Heidelberg. ISBN 978-3-540-11494-9.

Sadilek, D.A., 2010. *Test-Driven Language Modeling.* Ph.D. thesis, Humboldt-Universitaet zu Berlin, Berlin, Germany.

Sadilek, D.A., Wachsmuth, G., 2009. *Using Grammarware Languages to Define Operational Semantics of Modelled Languages.* In W. Aalst, J. Mylopoulos, N.M. Sadeh, M.J. Shaw, C. Szyperski, M. Oriol, B. Meyer (editors) *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, 348–356. Springer Berlin Heidelberg. ISBN 978-3-642-02571-6.

Sametinger, J., 1997. *Software Engineering with Reusable Components*. Springer-Verlag New York, Inc., New York, NY, USA. ISBN 3-540-62695-6.

Scheidgen, M., 2009. *Description of languages based on object-oriented meta-modelling*. Ph.D. thesis.

Specification and description language - real time (SDL), 2013. [Online]. Available: http://www.sdl-rt.org. [Accessed: Dez. 18, 2013].

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*. Addison-Wesley Longman, Amsterdam, 2nd revised edition (rev). edition. ISBN 0321331885.

Szyperski, C., 1997. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional. ISBN 0201178885.

Tan, L., Sokolsky, O., Lee, I., 2004. *Specification-based Testing with Linear Temporal Logic*. In D. Zhang, É. Grégoire, D. DeGroot (editors) *IRI*, 493–498. IEEE Systems, Man, and Cybernetics Society. ISBN 0-7803-8819-4.

Thomas, M., 2011. *Modellbasierte Testgenerierung für Web-Applikationen. Bachelor Thesis.*

Tretmans, J., 2004. *Model-based testing: Property checking for real. Keynote address at the International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices, Available http://www-sop.inria.fr/everest/events/cassis04 (downloaded on 11.12.2013)*.

Utting, M., Legeard, B., 2006. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0123725011.

Utting, M., Pretschner, A., Legeard, B., 2012. *A taxonomy of model-based testing approaches*. Software Testing, Verification and Reliability 22(5):297–312. ISSN 1099-1689.

Weißleder, S., 2010. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation from UML State Machines*. Ph.D. thesis, Humboldt-Universitaet zu Berlin, Berlin, Germany.

Woodcock, J., Davies, J., 1996. *Using Z: Specification, Refinement, and Proof (Prentice-Hall International Series in Computer Science)*. Prentice Hall. ISBN 9780139484728.

Zeng, H., Miao, H., Liu, J., 2007. *Specification-based Test Generation and Optimization Using Model Checking*. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, TASE '07, 349–355. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2856-2.

Zschau, J., Gasparini, P., Papadopoulos, G., Filangieri, A.R., Fleming, K., the SAFER Consortium, 2007. *SAFER - Seismic eArly warning For EuRope*. In *EGU General Assembly*. Vienna, Austria.

List of Figures

List of Tables

List of Listings

Definition Index

List of Abbreviations

ATN 103
The Abstract Test Notation (ATN) is a textual concrete syntax for the MBT metamodel described in this dissertation. With ATN, one can configure the test generation and write explicit test case specification in LTL. The Azmun test generation tool uses ATN also to represent the generated test cases.

CCM 18
Common Object Request Broker Architecture (CORBA) Component Model. The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together (i.e., it supports multiple platforms).

CMOF 29
The MOD standard is divided into the essential MOF and the complete MOF (CMOF).

COM 18
Component Object Model. The Component Object Model (COM) is a binary-interface standard for software components introduced by Microsoft in 1993. It is used to enable interprocess communication and dynamic object creation in a large range of programming languages.

CTL 85
In logic, computation tree logic (CTL) is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be an actual path that is realized..

Acknowledgements

Ich danke meinen Betreuern, Prof. Dr. Joachim Fischer und Prof. Dr. Holger Schlingloff, für ihre langjährige Unterstützung in allen Phasen meiner Promotion. Mein Dank gilt zudem meinen Kollegen des Graduiertenkollegs METRIK, vor allem Stephan Weißleder, Daniel Sadilek, Arif Wider, Dirk Fahland, Markus Scheidgen, Michael Frey, Andreas Dittrich und Frank Kühnlenz.

Ich möchte weiterhin meiner geliebten Ehefrau für ihre Unterstützung und Zuversicht während allen Phasen meiner Promotion, gerade aber ihrer Geduld während der Endphase, danken.

Ich widme diese Arbeit allen Absolventen einer Fachhochschule, die ihre Promotion anstreben.

Selbständigkeitserklärung

Hiermit erkläre ich, dass

- ich die vorliegende Dissertationsschrift „Model-based Testing of Dynamic Component Systems“ selbständig und ohne unerlaubte Hilfe angefertigt habe,
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist gemäß des Amtlichen Mitteilungsblattes Nr. 34/2006.

Berlin, den 14. Jan. 2014